UiO : Department of Informatics
University of Oslo

Implementation of a real–time distributed video processing pipeline

Sigurd Ljødal
Master's Thesis Autumn 2014

Implementation of a real–time distributed video processing pipeline

Sigurd Ljødal

Abstract

Video processing is a resource demanding task. While todays high–end machines are able to process and encode video at reasonable speeds, they are usually not capable of doing this in real–time.

In this thesis, we investigate and implement a distributed version of the real–time panorama video processing pipeline from the Bagadus project. The pipeline consists of several processing steps. Images are first captured from five individual cameras, and then grouped into sets. These sets are converted from the Bayer image format to YUV444 format, before they are stitched into a large panorama image. The stitched panorama video is encoded into H.264 format and stored on disk. It is also possible to enable HDR mode in the pipeline, which creates a video with more details visible in shadows and light areas.

We initially created a simple distribution setup, allowing individual processing steps to be run on separate machines. To improve the performance of this setup, we have implemented a more advanced setup. The improved setup removed bottlenecks and adds support for Nvidia GPUDirect, for minimal latency GPU–to–GPU memory copies between machines. This enables a large number of setups, with minimal delay added by the distribution.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisors Pål Halvorsen, Håkon Kvale Stensland, and Carsten Griwodz, who have been very helpful, providing feedback, advice, and support. I would also like to thank Ragnar Langseth and Asgeir Mortensen for the great cooperation and countless discussions.

In addition, I would like to thank Roy Nordstrøm, Hugo Kohmann, and the rest of the team at Dolphin Interconnect Solutions, for the great support, and Hermann Möhring for providing invaluable feedback and writing tips.

Finally, I would like to thank family and friends for continued support.

Oslo, August 15, 2014
Sigurd Ljødal

Chapter 1

Introduction

1.1 Background

Video processing system has the potential to change how we interact with and use video in our daily lives. Tasks commonly handled by manual labor today, can be automated: from fun hobby projects, like detecting squirrels on bird feeders [1], to large scale systems like goal detection technology [2]. As the systems get bigger, the resources required for processing increases and the system complexity grows. With todays cloud computing solutions, it is easy to run resource demanding processing and analysis, with resources allocated as needed. While this is perfect in many situations, the delays added by using off–site processing, and limited control over hardware, can be problematic for real–time applications. Real–time processing and analysis of video allows feedback to be provided almost instantly. The feedback can be applied in many different situations, from spraying water in the first example, to deciding, if a goal has been scored in the second example. This requires minimal delay, and it can therefore be hard or impossible to implement using cloud based solutions.

Such a real–time system, developed by the iAD Research project at the University of Oslo, is the Bagadus system. The system is built for analysis of arena sports, with the current implementation specialized for association football. It is focused around a panorama video of the entire playing field, which is combined with information from other sources to provide a powerful analytics system that requires minimal human interaction. With player tracking provided by ZXY [3], it is possible to automatically generate personalized video streams. Annotations gathered in real time from the coaching staff, are used to create videos of events. The videos are available for playback within seconds. In addition to manual annotations, events can also be automatically created by analysis of the player movements and positions. Video of the gathered events are available for replay almost immediately. One possible use case for this system is video replay during the half–time break. Another is immediate replay during practice, or event replays on stadium big–screens.

In the initial implementation, the panorama video was created from four cameras. This provides a good overview of the field, but the resolution and overall image quality is too low for zooming in on parts of the image. To increase the image quality, the cameras will be replaced by five new cameras, with higher resolution. In addition to the increased resolution, the new cameras can also deliver up to 50 frames per second, allowing research into new areas.

With the four original cameras, delivering up to 30 frames per second, the system runs in real–time on a single high–end machine, making extended use of Graphics Processing Units (GPU) to offload the Central Processing Unit (CPU). With higher resolution images and increased frame rate, more than three and a half times as much data is produced by the new cameras. This requires more processing resources, and it is unlikely that it will run in real–time on the existing hardware. Instead of investing in expensive hardware, we will investigate the possibilities of running the system distributed across multiple machines.

1.2 Problem Definition

Running the system across multiple machines requires extra considerations compared to parallel execution on a single machine. With a single machine, the memory is shared. In contrary, distributed systems usually consist of separate units, each with their own memory. This requires extra care to be taken when designing the system, as the usual tools for efficient parallelization like condition variables, mutual exclusion, and semaphores can be complex or impossible to implement with reasonable performance.

In this thesis, we will first explore and implement a distributed processing pipeline, not only capable of producing a panorama video from the five new cameras in real–time, but also capable of handling future extensions of the system. As the system is continuously developed, it is likely that processing steps will be added or removed. This should not require changes to the distribution code.

We will discuss how we are going to distribute the processing, including how to distribute the work–load and also how to control the program execution across machines. The distribution should preferably not place any limitations on the use of available system resources, and it should be as light–weight as possible.

Once a working prototype has been completed, we will focus on improving performance and minimizing delay. With much of the processing in our pipeline performed on GPUs, we will need to copy memory between GPUs in different machines. This can potentially be a bottleneck. We will investigate the possibility of using Nvidia GPUDirect [4] to optimize the data flow between GPUs.

1.3 Limitations

While the topic of this thesis is the panorama processing pipeline, our focus will solely be on the overall pipeline design and distributed processing. Other research topics like image quality have already been discussed in great detail in multiple research papers and master thesis's [5--9].

It is also likely that optimizations to the individual processing steps can be applied, but this will not affect the distribution of the system and is therefore outside the scope of this thesis.

1.4 Research Method

While working on this thesis, we have redesigned and reimplemented the existing Bagadus panorama processing pipeline prototype. The prototype implementation is working and in use at Alfheim Stadion in Tromsø, Norway. We have evaluated and tested our implementation. This corresponds to the *design paradigm* of the ACM classification [10].

Evaluations have been performed both in a real–life scenario, with our prototype setup at Alfheim Stadion, and at machines in our lab at the University of Oslo.

1.5 Main Contributions

In this thesis, we have shown that the Bagadus video processing pipeline efficiently can be distributed to work across multiple machines.

We first designed and implemented a common interface for communication between the different steps of our processing pipeline. By using this interface for all steps in the pipeline, we have created a pipeline where parts of the pipeline easily can be moved or reordered.

Next, we designed and implemented modules for transferring data between machines. These modules use the exact same interface as the processing steps, and behave like the processing steps. This allows them to be inserted anywhere in the pipeline.

When the distributed pipeline had been implemented, tested, and found to be working, we turned our attention to improve the distribution by utilizing technology like Nvidia GPUDirect and by further improve memory handling related to the distribution.

In addition to the panorama pipeline, we have also implemented other related tools, like a simple web interface for scheduling and starting recordings.

We have also published two papers while working on this thesis:

Bagadus: An Integrated Real-Time System for Soccer Analytics
: In this paper we introduce the new panorama image algorithm and the new distributed processing pipeline [9].

Interactive zoom and panning from live panoramic video
: This paper describes the virtual camera viewer created to play back the panorama video [11].

1.6 Outline

In chapter 2, we introduce Bagadus and all related work that together make up the complete system. This includes an introduction to the processing pipeline, which will be the topic of this thesis.

Next, in chapter 3, we discuss the pipeline in detail. Here we introduce the steps taken to prepare the pipeline for distribution, before moving on to the actual distribution process, with an evaluation of possible technologies and a reasoning of our choice. We will also look at how we can optimize the pipeline further.

In chapter 4 we evaluate the performance of our implementation. Both the complete pipeline, and individual parts like Nvidia GPUDirect RDMA are evaluated, to see which combinations return most in terms of performance and usability. Both various distribution setups and different hardware are tested.

A summary and conclusion is provided in chapter 5.

Chapter 2

The Bagadus system

Bagadus is a system created to help coaches and players analyze events during matches. It is built around a panorama video created from cameras covering the entire playing field. This panorama video is generated in real–time through a processing pipeline, which will be the topic of this thesis. The system has been developed in cooperation with Tromsø IL (TIL) [12]. We currently have a prototype setup at their home stadium; Alfheim Stadion in Tromsø, Norway.

2.1 Panorama video pipeline

The panorama video pipeline is the core of Bagadus. It captures video from cameras covering the entire playing field and creates a large panorama video. The pipeline handles all steps from image capture, to processing and stitching, before finally writing the panorama video to disk. From its initial stages as an offline stitcher, the pipeline has progressed into a real–time processing pipeline that creates panorama images on the fly. [13--15]

In this chapter, we will introduce steps taken to improve the processing pipeline. This includes new and better cameras, improved processing steps, and automated video recording. Starting from the existing setup, we will discuss steps taken to improve upon the existing pipeline. This is done to achieve better image quality and prepare for future expansion. Improvements includes better cameras, new algorithms for panorama image generation, and distribution.

2.1.1 Cameras and lenses

To improve the visual quality of the panorama video, we have upgraded the cameras. The old pipeline used four cameras [16] with a resolution of 1294×960 pixels and a maximum rate of 30 frames per second (fps). These cameras have been replaced by five new cameras [17], which have a resolution of 2046×1086 pixels[1] and can deliver 50 fps. This doubling of resolution and a greatly increased frame rate do not only increase the image quality, it also allows us to experiment with new technologies, such as high-dynamic-range imaging (HDR).

[1] Because of the sensor format of the cameras, the actual useable resolution of the output video is different from the sensor resolution. This results in the video resolution of the new cameras being 2040×1080 pixels.

In addition to the new cameras, we also have new lenses. With the old lenses, there was a fair amount of overlap between the images from each camera. This made the images useable independently, but it also reduced the effective resolution of the panorama video. To improve this, we use new lenses that give minimal overlap between the cameras. This is better suited for panorama generation, but also makes the individual images almost useless, as they only cover a narrow part of the field.

2.1.2 Panorama generation

Accompanying the new camera setup, is a new image stitching algorithm. The old panorama pipeline used a rectilinear projection, which creates a highly distorted images at wide field of views. This resulted in a very wide and narrow panorama, with visual stretching on each side of the image. In the new setup, the projection has been changed to a cylindrical projection. This is better suited for wide view angles. A rectilinear projection becomes very distorted at viewing angles above approximately 120°. A cylindrical panorama can be used for extreme viewing angles, even full 360° panoramas. The Bagadus panorama covers approximately 160°, and is thus more suited for a cylindrical panorama [8].

Figure 2.2 on page 9 shows the quality difference between a panorama image generated with the old and the new camera setup.

2.1.3 Processing steps

The processing steps in the pipeline include everything from format conversion, to stitching together the individual images to a large panorama image. The pipeline is divided into separate modules. Each module has a separate responsibility and operates independently from the other modules. A short introduction to each module is given here. For a more thorough introduction please see [8]. Figure 2.1 on the following page shows how the modules are connected together in the pipeline.

Camera module

The camera module is responsible for interacting with the camera drivers. Each frame is returned in a format readable by our processing pipeline. Each of the five cameras, used in our pipeline, are handled independently with one camera module for each camera.

This module is also responsible for calculating the timestamp when the image was captured. Each camera has an internal clock. This is not a real–time clock, and the clocks of each camera can be different. To get a timestamp for each frame, the camera module calculates a timestamp based on its own real–time clock. This is important, as we need to synchronize the frames from the individual cameras before creating the panorama image.

Frame synchronizer module

The frame synchronizer module receives frames from each individual camera. As each camera operates individually, it must make sure that the correct sets of frames are joined together. This is done based on the timestamps assigned to each frame by the camera modules.

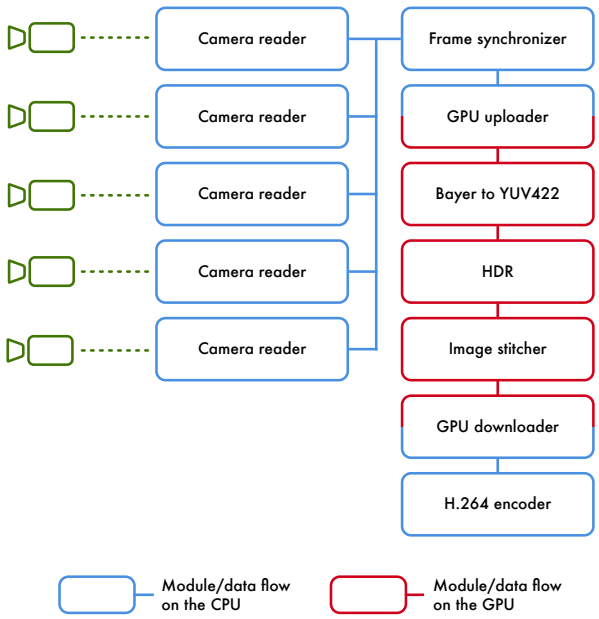


Figure 2.1: Panorama processing pipeline steps on a single machine.

For reasons out of our control, some frames from the cameras are dropped. It is the responsibility of the frame synchronizer to handle dropped frames. Without handling, dropped frames could create problems for our pipeline. To ensure that we keep a steady frame rate, dropped frames are handled by repeating the previous frame.

CUDA uploader and downloader modules

Most of the processing performed in our pipeline is done on GPUs, using the Nvidia CUDA library. Both the camera driver and the encoder run on CPU. Therefore we need to copy the frame data to GPU memory before processing can be performed, and back to CPU memory before encoding. To handle this, we have created two modules, one for copying to GPU memory and one to copy back to CPU memory.

Bayer to YUV converter

Each camera is connected to the computer with a single 1 Gbps ethernet connection. This limits the formats we can receive images in from the cameras. The old pipeline used the YUV420 image format. With the new cameras at 50 fps, this would require more than the available bandwidth.

Because of this, we capture images in Bayer format, which is the native format of the image sensors in the cameras. Bayer is a format where there is only one value for each pixel. This value is for either red, green, or blue, depending on which pixel it is.

While this format is very space efficient, it is not suited for the needs of our pipeline. We therefore convert from Bayer to YUV444 before performing any processing in the pipeline. This conversion is performed on the GPU.

Panorama stitcher

The panorama stitcher is responsible for stitching the five individual images into one large panorama. Compared to the old pipeline, this module has been rewritten using a cylindrical projection, instead of a rectilinear projection. The result is a panorama image with less stretching.

The stitching steps are performed on the GPU, and this module reads input frames in YUV444 format and produces output images in YUV422.

HDR module

During difficult lighting conditions, parts of the playing field can become hard to see. This particularly happens in the evening, when stadium buildings cast shadows onto the field as the sun is setting. Images, captured from the cameras under these conditions, can be very dark or very light in some areas. This can make it difficult to see details. To improve this, we have implemented a high–dynamic–range (HDR) image module for our pipeline [18].

An HDR image is a single image created from multiple images of the same subject, taken at different exposure times. The combined image can then contain more information in the shadows without loosing details in lighter areas.

In our pipeline, the HDR images are created from two separate images. One which shows details in shadows, and one which shows details in the light areas of the field.

H.264 encoding

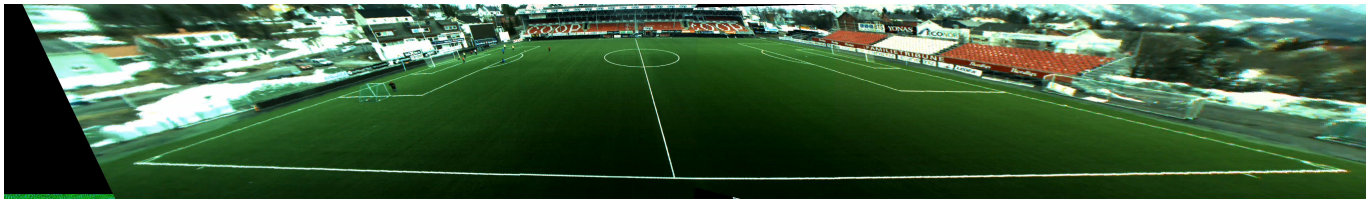
After processing, the panorama image is saved on disk. For practical reasons, the video is stored as a compressed H.264 stream. Compared to the old pipeline, the encoding performed here has been optimized. We have removed abstractions used in the old pipeline and now use the `x264` [19] library directly. This gives use more control over the encoding steps, which results in much more efficient encoding. The output files from the panorama pipeline has been reduced in size from 40– 50 MB to 5–10 MB per 3 seconds of video, without noticeable loss in quality. After the video has been stored on disk, it can be picked up for distribution to viewers via internet or to other parts of the system for further processing.

2.1.4 Increased workload

As a result of the increased frame rate and image size, the overall workload of the processing pipeline has increased.

With the cameras connected through ethernet to the machines, the frame retrieval generates lots of interrupts. In addition, the camera drivers use lots of threads. This results in many context switches, and uses valuable resources. As we use the `x264` video encoder library, which runs on CPU, the overall CPU load of the pipeline can be very high.

In the old setup, the four cameras were connected directly to a single processing machine. With five cameras, we no longer have enough ethernet ports to connect the cameras to this machine. Additionally, the work load has increased: the frame rate is increased, and the image resolution is higher. This means that the pipeline is unable to run on the current six–core machine.



a) Rectiliear panorama, 1k cameras

b) Cylindrical panorama, 1k cameras

c) Cylindrical panorama, 2k cameras

Figure 2.2: These images show the evolution of the panorama. (**a**) shows the original rectilinear panorama, captured with four 1k cameras. (**b**) shows how the new cylindrical panorama image looks using the old 1k cameras, and (**c**) shows the current output image from the pipeline with five 2k cameras. Please note that the difference between these images on paper are small. To better see the difference zoom in on the PDF.

We now had to make a choice. We could further upgrade the processing machine to make it able to still run the pipeline, or we could distribute the processing over multiple machines. Both have their benefits and drawbacks. Running the pipeline on a single machine reduces complexity and allows the processing steps to be interleaved in ways not possible when running across multiple machines. Unfortunately, it limits the scalability of the system. Distributing the system improves the scalability beyond what is possible on a single machine. It does, however, also increase system complexity and eliminates some alternatives.

We have chosen to distribute the system, as the benefits of distribution and being able to continue to use commodity hardware outweigh the downsides of running on multiple machines.

2.2 Automatic recording

We want to make the system as easy as possible to use. To achieve this, recordings should start and stop automatically without interaction from users. The only thing required from users is that they enter a schedule ahead of time. The pipeline has therefore been designed to operate autonomously. Once it has been started, no interaction with the pipeline is required. When a scheduled recording is coming up, the pipeline will automatically start recording.

2.2.1 Scheduling recordings

To schedule recordings, we use a database. This database stores all the information we need about each match, including start and stop times, which teams are playing, information about each team (like the players names and jersey numbers). It also stores what path the output panorama video files should be written to.

To enter information into this database, a web interface has been set up. This interface allows users to schedule recordings by entering a start and stop time. Users can also start an immediate recording of a given duration, with a set of easily accessible buttons. This interface is completely separate from the processing pipeline. The pipeline does not depend on this interface, and it can operate without it. Figure 2.3 on the next page shows the web interface for scheduling recordings. The interface also shows a list of already scheduled recordings.

In addition to manually scheduled recordings, we also use external sources to gather as much information as needed about teams, fields, and also scheduled matches for each field. These matches are automatically entered into the database, just like manual recordings. For our users, the result is that most matches do not need to be manually scheduled, and will automatically be recorded.

2.2.2 Starting and stopping recordings

Once the schedule has been saved in the database, we want the pipeline to automatically start and stop recording, based on the schedule. To achieve this, the pipeline must either be notified whenever it should start or stop a recording, or it must check the database.

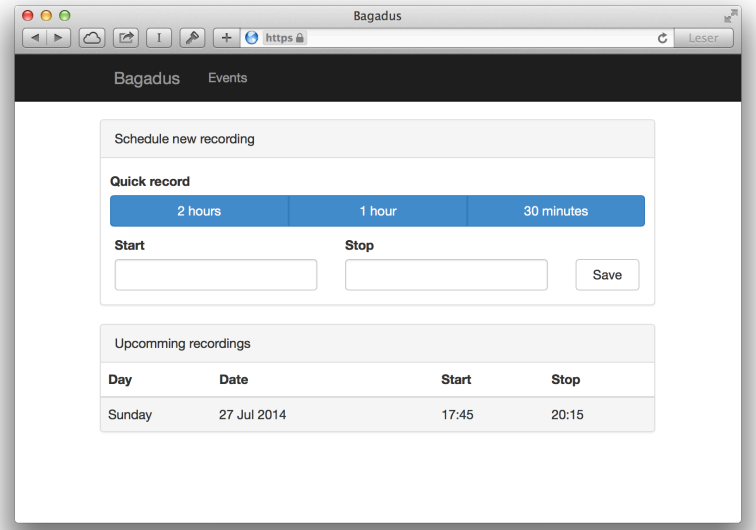


Figure 2.3: Interface for scheduling recordings

The first option could potentially simplify the pipeline greatly. We would not need any code handling schedule updates, and no connections to a database would be needed. The notifier could also start the pipeline manually whenever a recording should start. This has some downsides, too. It would require an additional component, that has to handle database connections and somehow communicate with the pipeline. We also risk the pipeline starting multiple times, if something goes wrong.

The second option adds complexity to the pipeline. It must take care of checking the database regularly, or, if possible, have the database notify the pipeline of any changes. As it becomes aware of when the next recording should start, it must make sure that a recording is started at the correct time. This reduces the number of components, as the pipeline can connect directly to the database. It also eliminates the chances of two pipelines getting started at the same time.

The old pipeline used the first option, where a new pipeline was started for each recording by a separate script. The application would sleep for a given amount of time, before starting to record, and then run for a given duration before exiting.

In the new pipeline, we have decided to go for the second option. It was implemented by having the pipeline connect to the database at regular intervals. Based on the schedule stored in the database, it will start and stop recording. In addition to this automatic database lookup, the pipeline can also be sent the `SIGHUP` signal. This tells the pipeline to immediately check the database. This is useful when it is desirable to start a recording with minimal delay.

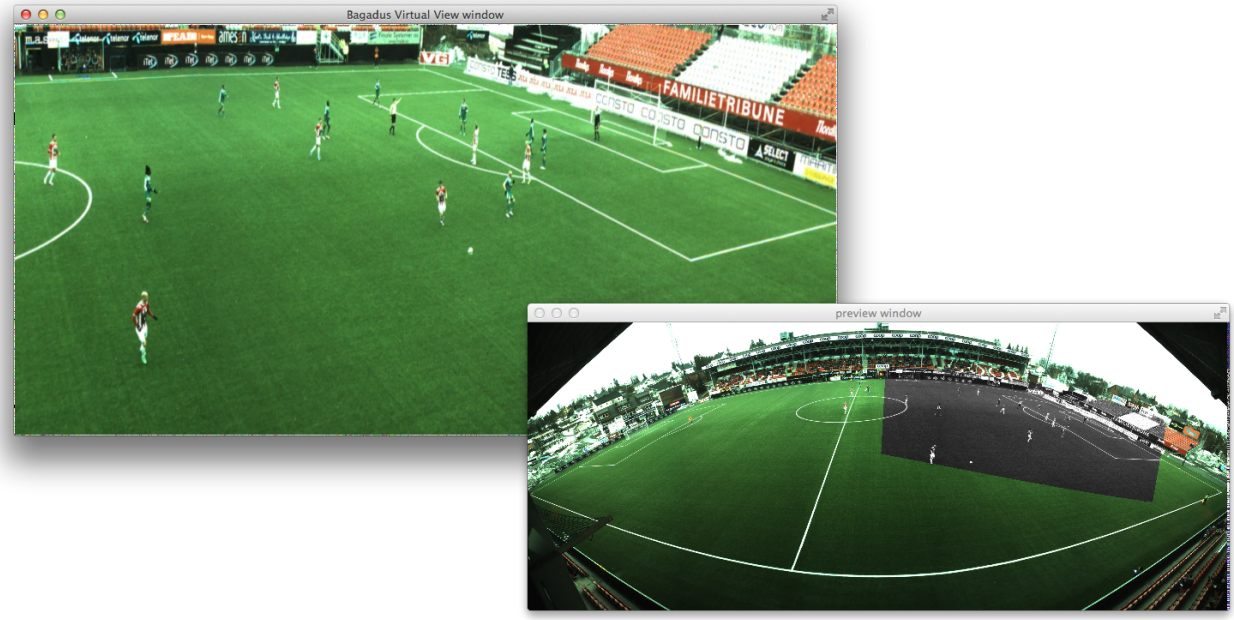


Figure 2.4: This image shows the virtual camera viewer. The larger window on the left shows the virtual camera, while the smaller one on the right shows the entire panorama video, with the section shown by the virtual camera marked in gray.

2.3 Other components of the system

We have also researched and developed other work, related to the Bagadus system and the panorama pipeline. This includes ways to play back the video and other research to further improve the usability of the system.

2.3.1 Virtual camera viewer

For watching recordings from the panorama pipeline, we have created a virtual camera viewer. This viewer acts as a virtual camera that allows panning and zooming in the panorama video, just like a camera man would zoom and pan during the match with a normal camera. Unlike a normal camera, the virtual camera is created from the panorama video. This allows zooming and panning to be done at any time. The viewer can be used for both live viewing and for viewing recordings.

As the entire field is captured, users can choose what they want to view independently. Moving the virtual camera can be done manually, or automatically based on input from various sources. With tracking of the players, the camera can be moved automatically after a selection of players. If the ball is being tracked, this information can also be used automatically move the virtual camera.

For a preview of the viewer, see figure 2.4 above.

2.3.2 Player tracking

In addition to the panorama video pipeline, another important part of the system is the player tracking, provided by ZXY Sport Tracking AS [3]. This system tracks player positions[2]. When we combine this information with our panorama video and the virtual camera viewer, we can offer highly customized viewing experiences.

In addition to plainly following players, the tracking information can also be analyzed to automatically extract interesting parts of the video. Examples of events that can be extracted from player tracking, are all situations where a player is within the opposite team's penalty area, or situations where many players are moving quickly from one side of the field to the other.

The player tracking information is used to automatically move the virtual camera based on player movements.

2.3.3 Ball tracking

We have researched the possibility of visually tracking the ball in the panorama image. The position of the ball could potentially be very useful. It would allow the virtual camera to automatically follow the ball around the field. Additionally, it could be used for automatic analysis of the situation on the field.

This research is currently at an early stage. We have implemented a prototype version of the virtual camera viewer which automatically can follow an object. This work is being built on to allow tracking of the ball. We have compared different algorithms and evaluated how these perform in different situations [20].

2.3.4 In–browser virtual camera viewer

The current virtual camera viewer is implemented as an application that performs processing on GPUs using CUDA. This is an unfortunate dependency, and it limits the possible uses of the viewer. To improve the usability of the viewer, we have researched the possibility of rendering the video on a server. The video could then be delivered to to clients as a pre–rendered video. This removes the hardware dependencies, and also has the benefit of only requiring users to have a browser to view the videos.

The viewer is run on a server, just like the normal viewer would be run at any machine. Instead of rendering the generated video to a screen, the buffers are passed to the hardware encoder on the GPU and directly encoded into an H.264 stream. This ensures minimal latency. Movement input is transferred from the browser to the server through web sockets.

As part of the research, we have also evaluated different H.264 encoder implementations, including software and hardware implementations. This shows that hardware encoders offer comparable quality to software encoders, with minimal resource usage [21, 22].

[2]The ZXY systems also track vitals like heart rate. This can be used for overlays or event extraction, but it is not currently used by Bagadus.

2.3.5 Event tagging

In addition to the automatic events, created from sources like player tracking and external sources, a smart phone application, called Muithu [23, 24], has been developed. It allows the coach or other experts to annotate events live during the match. This includes marking events like offside, foal play, and similar events that can be hard to detect automatically.

Based on the information gathered through the app, we can present the user with a playlist of events. The events can be automatically filtered based on which player is viewing. Further, the video can be shown as a full panorama, or through the virtual camera viewer. A pre–generated virtual camera video can also be created, automatically following players selected in the event when combined with the player tracking system.

This system allows efficient and low–latency viewing of data. This is a great improvement over manually gathering and processing video from the match. Unlike manual systems, video clips based on events from the smart phone app can be made available for viewing within seconds of the user creating the event. For example, the system can be used during the half–time break.

2.3.6 Event viewer interface

Figure 2.5 on the next page shows an interface for viewing matches and events. It allows the user to select players and see them highlighted in the video. Under the video, an interactive timeline allows the user to move around in the recording. The timeline also shows annotations added by the coach.

We created this interface before the virtual camera viewer was implemented. Instead, it allows the user to switch between each of the cameras, either manually by clicking a button, or automatically based on the selected players.

At the bottom, the player overview can be seen. It shows the field from above with markers for each player. The players are color marked, based on which team they belong to, and selected players are highlighted.

2.3.7 Summary

As we have described, the Bagadus system has been greatly improved. We have upgraded the cameras to new, higher resolution ones, changed the algorithm used to generate the panorama image, and made other small improvements. We have also introduced the individual components, that together with the panorama pipeline make up the Bagadus system.

While the system has grown, the panorama processing pipeline is still limited to running on a single machine. To ensure stable processing of the current setup, and also to ensure scaleability beyond the current setup, without requiring expensive hardware, we want to distribute the processing across multiple machines. By distributing the load across several machines, we can continue to use commodity hardware, while delivering a high quality panorama video. This will be discussed in the next chapter. A prototype will be implemented, showcasing what can be achieved.

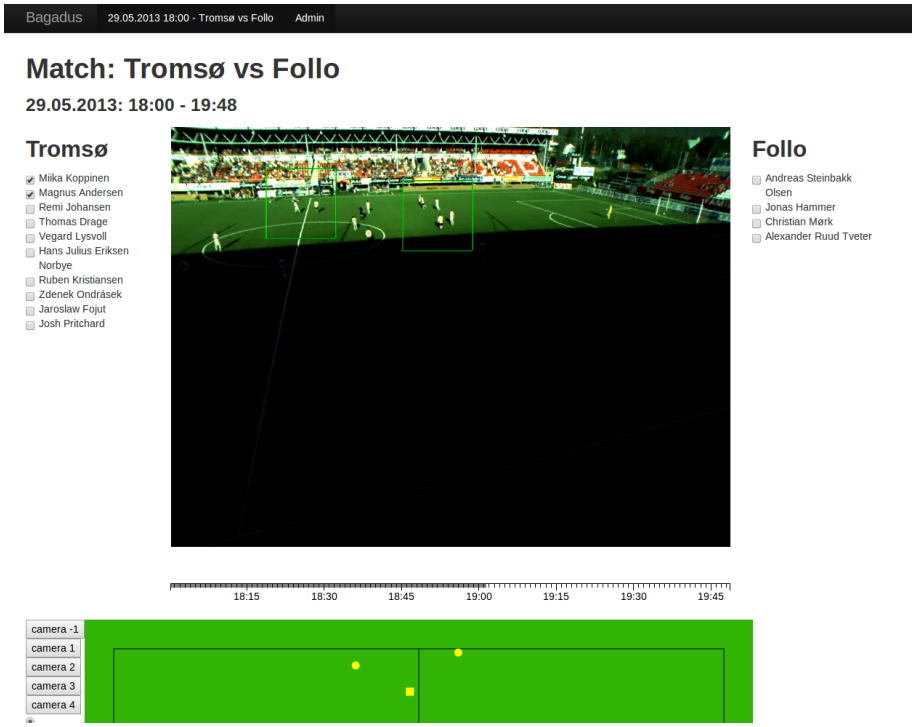


Figure 2.5: Match viewer interface.

Chapter 3

Distributed pipeline

The processing pipeline handles everything from capturing images from the cameras, to writing the finished panorama video to disk. It has gone through substantial changes during the last year. From running on a single machine with four cameras, it is now running across multiple machines with five higher resolution cameras.

In this chapter, we will introduce the pipeline in detail. We have redesigned and reimplemented the most of the pipeline and will discuss the changes we have done. We start with the steps taken to prepare the pipeline for distribution. We then move on to the actual distribution of the pipeline. In the end, we discuss optimizations we have applied to the distributed pipeline.

3.1 Module design

Starting almost from scratch, we have redesigned and reimplemented major parts of the pipeline. The old pipeline was heavily optimized for one specific configuration, on a specific machine. While this allows assumptions to be made about the environment, it also makes the pipeline less portable and harder to extend. To make it easier to further improve the pipeline, we need a flexible pipeline, where new processing steps can be added, and old ones removed.

Embracing the pipeline design, we have decided to make each step a separate module. Each module should be designed to perform a specific task. Modules should be independent, and not have dependencies on other modules. By following these restrictions, we can create a pipeline where individual modules can be moved around or reused with ease.

The first thing we need, is an interface for communication between the modules in the pipeline. Modules should be able to control the data flow between them, without a third party interfering or having to supervise the process. To design this interface, we first looked at what we needed in the pipeline and grouped the processing steps into categories. We found that we have three major kinds of processing steps: steps that produce an output, steps that consume an input, and steps that both consume and produce.

Producer

Producers are processing steps that create an output, that can be used for further processing. In our case this output is a frame. An example of such a step in our pipeline is the camera reader module. It reads frames from a camera, and outputs them for further processing in the pipeline.

Consumer

Consumers are processing steps that takes an input, and processes it. The output from these modules are generally not useable in other processing steps. The H.264 encoder is an example of a consumer processing step. It encodes each frame it receives into an H.264 stream. The resulting H.264 stream is saved to disk.

Combinations

Most of our processing steps are a combination of the two types described above. They get their input from the previous processing step and give their output to the next processing step. Examples of processing steps that behave like this, are the de–bayering step, the frame synchronizer, and the image stitcher.

3.1.1 Module interface

The pipeline will always start with a producer module. This module will usually be the camera reader module. It passes the captured frames to the frame synchronizer, which again passes the frame set down the pipeline for processing. In the previous pipeline, this is handled by a global buffer handler. This handler pushes the frames down the pipeline each time a new frame has arrived. This is a simple solution, and it allows a fine grained control over buffers. It does, however, also increase the processing latency per frame. At 50 fps, each step will take exactly 20 ms. If the processing of a frame takes less than that, the frame will be waiting for other processing steps to complete, before being passed on to the next module. If processing takes more than 20 ms, the frame will be dropped. This is unfortunate. We want to minimize the processing latency, while being flexible, to allow for spikes in processing times. To do this, we have looked into two different design patterns: push and fetch.

Push

Push is a design where each module is responsible for delivering frames it has processed directly to the next module in the pipeline. This ensures minimal delays, as the next module is immediately notified when a frame is ready for processing.

Fetch

The fetch design requires modules in the pipeline to ask the previous module for a frame. While this can eliminate delays compared to the global buffer handler, it is not guaranteed to be as efficient as a push based design.

Both of these patterns have their own strengths and drawbacks. A push based pipeline would eliminate the delays, but it could make our modules more complex. As the modules can possibly receive more frames than they can process, modules would be required to cache frames. With a pull based pipeline, each module have full control of their buffers and will never be given more frames than it can handle. This difference makes pull based modules less complex, but more care is required to minimize latency.

```
class FetchModule {
    public:
        virtual struct header *getFrame(struct header *f=NULL) = 0;

        // Get a frame in Cuda memory.
        virtual struct header *getCudaFrame(struct header *f=NULL) = 0;

        // Get a frame in a Cuda array
        virtual struct header *getCudaArrayFrame(struct header *f=NULL) = 0;
};

class PushModule {
    public:
        virtual void putFrame(struct header *f, int id) = 0;
};
```

Figure 3.1: Module interface

One strength of the pull based layout is that it, with some limitations, makes it easier to avoid unnecessary copies of the frame buffers. In the method call to request a frame, the caller can pass in a pointer to a buffer. The callee can then write its output directly to this buffer. A similar performance gain can be achieved in push based modules by using a pre-allocated buffer pool, but this is more complex than in a pull based design. In a pull based pipeline, this design pattern does, however, have one significant drawback. The request for a frame must be called before the previous module in the pipeline can perform any operations. To minimize or even eliminate this issue, the modules can be implemented with two buffers. One is used in a call to the previous module. The other is used for processing as soon as the next module has asked for a frame.

We decided that most of our modules would work perfectly with a pull based interface. However, we realized that some modules would benefit greatly from having a push based interface. This specifically applied to the frame synchronizer, which receives frames from five different modules. Instead of it requesting frames from five other modules, we decided to also design an interface for push based modules, which can be used by the frame synchronizer.

The interface we ended up with has changed slightly throughout the last year, but the basics are still the same. The interface is implemented as an abstract C++ class, with methods that all modules should implement. The interface currently in use, can be seen in figure 3.1 above. This interface has two main classes. `FetchModule` is an interface for modules that frames can be fetched from. `PushModule` is an interface for modules that frames can be pushed to. At present, the only module using the push based interface is the frame synchronizer. All other modules use the fetch based interface.

The `FetchModule` interface has three different versions of the `getFrame` method. The first version is the method you call if you wish to get the frame in a normal CPU buffer. The second, `getCudaFrame`, returns the frame in a normal buffer on the GPU. The last version, `getCudaArrayFrame`, also returns the frame in GPU memory, but this time it is in a CUDA array buffer. To decrease the complexity of the modules, most modules do not implement all of these methods.

```
1  extern "C" struct header
2  {
3      struct timeval  timestamp;
4      uint32_t        frameNum;
5
6      int32_t         expoFirst;
7      int32_t         expoSecond;
8
9      uint32_t        flags;
10
11     uint32_t        size;
12 } __attribute__ ((packed));
```

Figure 3.2: C-struct for storing frame meta data

The FetchModule interface can be used in multiple ways. It is possible to call getFrame without an argument. This tells the callee that it should allocate a new buffer and return the frame in that buffer. It will then be the callers responsibility to clean up the memory afterwards. Alternatively, getFrame can be called with a pre-allocated buffer as an argument. This way, the callee will not have to allocate any memory, and should just copy its data into the pre-allocated buffer. The latter is the normal way our modules operate, as the overhead of allocating memory can potentially be a bottleneck [25].

The return value of a call to getFrame is also important. If something goes wrong, or we have reached the end of a recording, a NULL pointer is returned. When this happens, the module is expected to gracefully shut down. This involves cleaning up all resources it holds, return a NULL-pointer to the next module in the pipeline, and then exit. When no errors occur and we have not reached the end of a recording, the return value of the getFrame call will be a pointer to a frame. If a pointer to a frame was passed as an argument to the method, this pointer will be returned. When no argument is supplied, the return value is a pointer to a newly allocated frame.

We have not specified the details of how the modules should perform the processing. We only require that the family of getFrame calls should block until a frame is ready to be returned, but how that is handled, is unspecified. This means that when this method is called, it will not return until a frame has been received or an error has occurred. By convention, most modules wait until getFrame as been called before doing any processing. Because of this, modules should make sure to call it as soon as possible, and keep the delay between calls as little as possible.

3.1.2 In-memory frame meta data

In the common interface between modules, we use the C–struct defined in figure 3.2 above to pass information about frames. This contains all the information needed throughout the pipeline. Each of the fields are described below.

timestamp

This represents the moment when the image was taken. It is used several places in the pipeline, starting with the frame synchronizer. The synchronizer uses the timestamps of

the frames to group them into frame sets. The timestamp is also used in the encoder to make sure that the files are named correctly, based on which frames they contain. This is important as the timestamp in the filename is used for synchronization with the player tracking system.

frameNum
: This is simply a counter of the frames since the start of the recording. It is not currently in use in the pipeline, but it could become useful, and is therefore included.

expoFirst, expoSecond
: These values tell the exposure times of the frames. This is currently only used for generating HDR images, but they can also be useful for e.g. color correction. When running in HDR mode, a frame is actually a set of two frames. The `expoFirst` value is the exposure time used for the first frame, and the `expoSecond` is the exposure time for the second frame.

flags
: The `flags` field is used for storing general information about the frame. This field is used to indicate whether a frame is dropped or not, and it is used to identify if the frame is located in GPU or CPU memory.

size
: This stores the size of the frame. When running in HDR mode, the actual size of the frame buffer will be two times this value, as there are two frames in the buffer.

3.2 Distributed processing

As we said in the problem definition (section 1.2 on page 2), we want to distribute the panorama processing pipeline. As a part of this research, we have looked into several models for distributed processing, and various implementations of distributed processing frameworks. In this section, we will very briefly discuss these, before describing the solution we have chosen.

3.2.1 MapReduce

There are several models available for distributed processing. Examples include MapReduce [26] and Dryad [27]. MapReduce is one of the most common ways of processing large amounts of data in a distributed way. It consists of two, or sometimes three [28] steps. The data is first split into groups, also called *map*. These groups are processed individually. Next, these groups are *reduced* to smaller groups. In some implementations, a third step, called *merge*, is applied at the end. This allows post processing of the results.

One of the most common implementations of MapReduce is Apache Hadoop [29]. Hadoop is a collection of tools, which include an implementation of MapReduce for running on clusters of commodity hardware. There are also other implementations of MapReduce, like a multi–core version [30], a version for GPUs [31], and even for the Cell BE architecture [32].

While not initially designed for processing in real–time, MapReduce and Hadoop has now reached a point where it can be used for real–time processing. Facebook [33] has for example shown how they have used Hadoop for real–time processing in their messaging service [34].

There is, however, an important issue with Hadoop and the MapReduce design: these systems are designed for *batch* processing. Batch processing is a process where a large set of data is split into smaller chunks. The chunks are processed independently, and only after processing the data is joined together to produce on large output.

This is not a programming model that is suited for our workload. For efficient processing with MapReduce, the data must be available before the program is started. We receive frames from the cameras as one continuous stream of data, and the data is received in real–time from the cameras. This means that no data is available beforehand. In addition, our workload is hard to split into chunks. While some of the steps can be performed on blocks of the image, other steps require access to the entire image. This is not possible or efficient with MapReduce. As we process the data in real–time, it is also important that we process the data in a near constant time. With large distribution systems like MapReduce, controlling the processing latency can be hard. Finally, we run each frame through multiple processing steps, where each step depends on the output from the previous step. This is called cyclic or iterative processing, and it is not possible with MapReduce.

3.2.2 P2G framework

To solve the problem of distributed real–time processing, like video processing, Beskow et al. [35,36] have demonstrated a processing framework called P2G. This is a framework created for arbitarily complex processing, created with real–time processing in mind. The P2G framework is based on experiences from the implementation of Nornir [37], which again is an implementation of a Kahn Process Network [38,39] (KPN). Compared to MapReduce, KPN is more flexible [40], but implementing a distributed KPN framework is more complex.

The P2G framework is designed specifically for real–time distributed processing of video. The framework also enables usage of heterogeneous resources, like GPUs. Unfortunately, this project is a research project, and the framework is not ready for production use.

3.2.3 Our solution

Without the possibility to use a distribution framework, we need to look into a custom solution. It is desirable to use a solution that requires minimal changes to the existing code. We know that our workload is iterative. Each frame must pass through each step, and most processing steps must be performed sequentially. Combined with large amounts of data passed between the processing steps, this creates a distribution problem where efficiency in the distribution is important. For example, at 50 fps, the data flow between the de–bayering module and the image stitcher module is greater than 2 GB/s.

Fortunately, each of the individual modules in the processing pipeline is capable of running in real–time on a single machine. This removes the need to distribute the individual processing steps across multiple machines. Because of this, we can distribute the processing in a very simple way. By running the individual processing steps on separate machines, we can distribute the load without requiring extensive changes to the code. Using the common module interface, which we

Frame rate	RGBA / YUV444	YUV422	YUV420	Bayer
25 fps	1762.56	881.28	660.96	440.64
30 fps	2115.07	1057.54	793.15	528.77
40 fps	2820.10	1410.05	1057.54	705.02
50 fps	3525.12	1762.56	1321.92	881.28

Table 3.1: Required bandwidth per camera for different image formats at different frame rates. This includes only the raw image data, and no overhead. The required bandwidth for YUV444 images in our pipeline is equal to RGBA bandwidth, as each value is padded from 3 bytes to 4 bytes for alignment. All values are in Mbps.

defined in the previous section, we can create modules that transfer frames from one machine to the next. This will require two modules, one that is a consumer module. This runs at the end of the pipeline on one machine, and transfers the frames to the next machine. On the next machine, there is another module. This module acts as a producer module. Frames from the first machine are received and delivered to the next module for processing.

This setup has several benefits. It requires minimal or no changes to the processing modules, as the distribution is taken care of without these modules' involvement, data transfers between the machines are kept to a minimum, and frames are only transferred at certain points in the pipeline and never more than once. There is, however, one important drawback. This setup does not facilitate distributing single processing steps over multiple machines. While this is still possible, it will require separate handling. As mentioned above, this is not a dependency on the current scale, and it is not likely to be necessary anytime soon. We will therefore not consider this scenario in this thesis.

Unfortunately, we have not been able to find scientific research into a distribution model like the one we are describing here. We will, however, explain and evaluate our implementation thoroughly, to ensure optimal performance.

3.3 Interconnect technology

When distributing the processing pipeline, we need an interconnect between the machines with high bandwidth. As mentioned previously, we need more than 2 GB/s at certain points in the pipeline. We therefore need a special interconnect solution to connect our machines into a cluster.

The large amounts of data eliminates using common 1 GbE connections, as they do not have the required bandwidth we need to transfer the frames between machines. Table 3.1 above shows the bandwidth needed per camera stream, for different formats and resolutions.

The pipeline uses a combination of Bayer, YUV444, and YUV422 between the modules. This results in different bandwidth requirements between different modules. Between the cameras, the frame synchronizer, and the de–bayering module, Bayer is used. With five cameras at 50 fps, the required bandwidth here is 4406 Mbps. Between the frame de–bayering module, the HDR

module and the image stitcher module, YUV444 is used. This requires 17626 Mbps. Between the image stitcher and the encoder, YUV422 is used. Here the image has been reduced in size, because of the stitching. The resulting bandwidth requirement is 5505 Mbps.

3.3.1 Ethernet

One alternative is to use 10 or 40 gigabit ethernet (GbE). The bandwidth with 10 GbE is enough to be able to transfer the data between the machines, as long as the pipeline is not split between de–bayering, HDR and image stitching modules. With 40 GbE, the bandwidth is high enough to split the pipeline anywhere. Compared to earlier standards like 1 GbE, 10 and 40 GbE offer higher bandwidth and lower latency.

3.3.2 Dolphin Interconnect Solutions

Dolphin Interconnect Solutions [41] (Dolphin) is an Oslo based company. It produces interconnect solutions with high bandwidth and minimal latency. Their products are based on the PCI Express (PCIe) standard, and use the PCIe protocol for communication between devices. To program for their products, Dolphin have one low level Application Programming Interface (API), the SISCI API, and one higher level API, called SuperSockets.

SISCI is an abbreviation for Software Infrastructure for Shared-Memory Cluster Interconnects and is an API developed in a shared European research program [42]. It offers low level access to the functionality of the interconnect cards. The functionality offered includes mapping of remote memory into local memory, remote interrupts, Remote Direct Memory Access (RDMA), and more. Application–to–application transfers of one byte across the cluster can be performed in less than 1 µs. The application–to–application bandwidth is above 3 GB/s.

SuperSockets is an implementation of the Berkeley Sockets API [43] over the PCI Express connection. This implementation allows programs to transfer data over the PCIe connection, using familiar Internet Protocol (IP)/socket based APIs. SuperSockets can be utilized in two ways. The SuperSockets library can transparently take over any sockets created in a program. This enables fallback to communicating over ethernet, if the PCIe connection is unavailable. Alternatively, SuperSockets can be explicitly used, through a new socket type defined by the library. The first alternative is preferred, as it requires zero change to existing programs, and also offers fallback to using ethernet in the case that PCI Express connection is unavailable.

Compared to 10 GbE, SuperSockets offer higher bandwidth and lower latency. With error checking build in to the PCIe protocol, there is no need for higher level error checking, like in Transmission Control Protocol (TCP).

In addition to being programmed explicitly, the Dolphin products can also operate as PCIe bridges, allowing other PCIe devices to communicate across the cluster transparently.

3.3.3 InfiniBand

Infiniband [44] is an alternative interconnect solution. Infiniband is a co–operation between multiple vendors, the two largest being Mellanox [45] and Intel [46].

	Dolphin IX	Infiniband	10 GbE
Bandwidth	40 Gb/s	100 Gb/s	10 Gb/s
Latency	below 1 µs	1 µs [1]	5–10 µs
IP/Sockets	yes	yes	yes
Remote interrupts	yes	yes	no
RDMA	yes	yes	yes [2]
GPUDirect	yes [3]	yes	no
Remote mapped memory	yes	no [1]	no

[1] Unknown/unconfirmed

[2] iWARP. Not supported on all 10 GbE equipment.

[3] We have developed GPUDirect support for Dolphin IX in cooperation with Dolphin.

Table 3.2: Interconnect features for different products.

The feature set, offered by Infiniband, is similar to the feature set offered by Dolphin. High bandwidth and low latency is the key here, too. RDMA, remote interrupts, and IP communication over Infiniband are available. Infiniband uses IPv6 for communication between devices. It is therefore not possible to use Infiniband products as transparent PCIe bridges.

The Infiniband products can be programmed with multiple APIs. Each vendor usually supplies its own API. In addition, a common low level API, called verbs, is available.

3.3.4 Feature comparison

Each of the interconnect solutions has its benefits and drawbacks. In table 3.2 above we have listed features that are useful for distributing our processing pipeline.

As we can see, 10 GbE offer a limited set of features. While the bandwidth is enough to support our pipeline at the current scale, it is not guaranteed to be able to scale. 40 GbE offer similar features to 10 GbE, but with higher bandwidth. This allows further scaling, but the ethernet standard is missing many features that would be practical in our pipeline.

Dolphin IX has lots of features and good performance. Compared to 10 GbE the bandwidth is higher and the latency is lower. In addition to the features listed in this table, transparent operation as a PCIe bridge is also possible. As mentioned above, this allows other PCIe devices to transparently communicate over the PCIe connectin. Dolphin is also working on 8 and 16 lane PCIe Gen3 products, that will offer higher bandwidth [1]. This removes the current lead Infiniband has, when it comes to bandwidth.

Infiniband offers almost all the features we want. It offers higher bandwidth than what is possible with Dolphin IX at the moment. Much of the same features are available, except for the possibility to operate as a PCIe bridge, and it is not possible to map remote memory.

Table 3.3 above shows a price comparison between Infiniband and Dolphin IX for a three node setup. This shows that the Infiniband solution is the cheapest of the two setups compared, but the Dolphin solution will likely offer higher performance.

We have chosen to go for Dolphin IX products. This is based on both the features available, and because that the University of Oslo has a great cooperation with Dolphin. This cooperation means that we get direct access to developers at Dolphin and great support.

[1] Gen3 products from Dolphin will be available in the autumn 2014.

	Dolphin IX	Infiniband
8P-switch	4990	1862.65
3x 1m cables	3×125	3×57.72
3x PCIe 8x cards	3×675	3×788.07
Total	7390	4400.02

Table 3.3: Interconnect pricing for a three node cluster. Prices for Dolphin IX equipment are collected from Dolphin, the switch is the IXS600, the cards are IXH610, and the cable is a 1m compatible copper cable. Prices for Infiniband equipment are from [47]. The switch is an Mellanox MIS5022Q-1BFR [48] 8–port 40 Gb/s switch, the cards are Mellanox MCB191A-FCAT [49] 56 Gb/s 8x PCIe Gen3 cards, and the cables are Mellanox MC2206130-001 [50] 40 Gb/s 1m copper cables. Please note that we do not have any experience with Infiniband, and do not know whether these Infiniband products are compatible. All prices are in US dollars.

3.4 SISCI API

The SISCI API is a low–level API, that allows access to all the functionality of the Dolphin IX cards. The entire API is accessible in user space, without requiring special privileges.

3.4.1 Segments

The most fundamental function of the SISCI API is to access memory in a remote machine. To facilitate this, there are a couple of requirements that must be fulfilled. In order to keep the delay low and minimize the processing requirements, memory accesses are handled directly by the PCIe card. These accesses do not go through the CPU. Modern computers use virtual memory addressing in user space processes, and the physical location of that memory is not guaranteed to be the same between accesses. Normally, when accessing memory through the CPU, the translation between virtual and physical addresses is handled transparently by the Memory Management Unit (MMU). When accessing memory from the PCIe card, we do not have access to the MMU. Because of this, we must access the memory with physical addresses. We must also make sure that the memory never is moved or swapped to disk. This is done by *pinning* memory. Pinned memory is handled differently by the OS, and it will not be moved or swapped to disk. This means that it safely can be accessed by using physical addresses.

In context of the SISICI API, memory is used through *segments*. A segment is a handle that, among other things, wraps a memory address and the size of memory allocated at that address. Multiple segments can be created on each node, so they are identified by an id. The id must be unique on the machine where the segment is allocated. This allows the segment to be uniquely identified in the cluster with the combination of the node id and the segment id.

By default, when a segment is created, pinned memory is allocated and associated with the segment. This memory resides in main memory, but it is not mapped into the application's virtual address space. This must be done manually. Segments can also be created without allocating memory. A segment created this way, does not have any memory associated with it by default. Using one of two, already allocated memory can be associated with the segment.

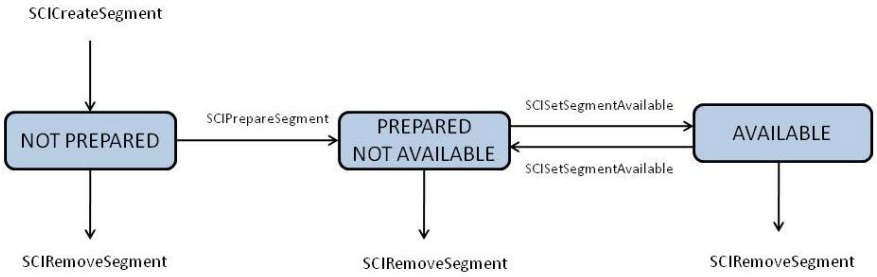


Figure 3.3: Diagram showing the various states a SISCI segment can be in [51].

The first function allows memory that is mapped into the virtual address space of the process, e.g. memory allocated with `malloc`, to be associated with a segment. The other allows memory that is not accessible in the virtual address space of the application, to be associated with a segment. This is usually memory that is located on a physical device, like a graphics card or a PCIe storage device.

Before a segment can be used by an PCIe adapter in the machine, it must be associated with the adapter. This associates the segment with the specified adapter, and prepares the segment for connection. After a segment has been associated with an adapter, it can be used in multiple ways. It can be made available for connections. This makes the segment visible for other machines in the cluster, and allows connections to be made from them. Segments can also be used privately, as sources for DMA transfers. This does not require the segment to be connectable, and therefore the segment does not need to have an unique id.

The various states of a SISCI diagram are shown in figure 3.3 above.

3.4.2 Connections and data transfers

When a segment has been made available for connections, other machines in the cluster can connect to it. Remote segments can be used in two ways. Either directly as a target for a DMA transfer from a local segment, or by mapping the segment into the application's virtual address space.

DMA requests are data transfers handled directly by a DMA engine on the PCIe card itself. This allows DMA transfers to be performed asynchronously, while the CPU performs other work. To perform a DMA transfer, one or more requests must be initialized and programmed to the DMA engine on the card. This adds a certain amount of overhead. As a result, for small transfers, the overhead can be a significant part of the time spent.

If the transfer sizes are small, mapping the segment into local address space can be quicker than using DMA request. This allows the processor to access this memory just like it would access local memory. This is generally most useful for smaller transfers where the overhead of a DMA request is too

Table 3.4 and 3.5 on the following page show bandwidth and latency for transfers with DMA requests and PIO on mapped memory. As we can see, the overhead of starting a DMA transfer makes it potentially slower. One of the major benefits of DMA requests over PIO is that the transfer can be performed asynchronously while the CPU is doing other work. For instance, this allows a transfer to occur while the data for the next transfer is processed. Since DMA transfers are handled by the PCIe card, the memory that we are copying from, must be pinned. This can add overhead, if it requires the data to be copied from another memory location and into the pinned segment memory. In such cases, mapping remote memory and performing the copy directly, using the CPU, can be faster.

Another possibility through the SISCI API, is broadcast transfers. With broadcast transfers, data can be written to multiple receivers simultaneously. The broadcast occurs at the switch level, so it is a true broadcast where the data is sent once and received multiple times.

Segments can also be shared by multiple applications on the same machine. To allow this, the program that created the segment must call `SCIShareSegment` to make the segment available for connections from other applications. After the segment has been made available, other applications can use `SCIAttachLocalSegment` to connect to a shared segment. There are no access restrictions on shared segments, so all connected applications have the same access to the segment.

3.4.3 Remote interrupts

Another feature of the SISCI API is remote interrupts. Remote interrupts work like normal interrupts. They allow a suspended thread on one machine to be waked up from other machines. Like segments, interrupts are identified with a per– machine unique id. Unlike segments, remote interrupts are always available for connections from other machines in the cluster.

Interrupts can behave in two different ways, depending on the options specified at creation time. In the default mode, the interrupt handler is not guaranteed to be called for each time an interrupt was triggered, if the interrupt is triggered repeatedly in a short timespan. When this happens, the interrupt handler is guaranteed to be called after the first interrupt was triggered and after the last one. The alternative mode guarantees that the interrupt handler will be called for each time the interrupt is triggered.

3.4.4 Other functionality

All accesses to segments can be synchronized by using the `SCIStoreBarrier` function, which blocks the caller until all write operations have completed. This can be used to ensure that a mapping is up to date before performing further processing. It is also possible to flush buffers with the `SCIFlush` function.

Lock segments are also available. Lock segments allow atomic test-and-set of a value. This can be used to create synchronization primitives like locks and barriers across machines.

Access to low level control/set registers (CSR) are also allowed through a series of functions. These allow reading and writing to local, or remote, CSR registers.

Transfer size (bytes)	Latency (µS)	Bandwidth (MB/s)
64	20.48	3.13
128	19.74	6.48
256	18.69	13.70
512	18.64	27.47
1024	20.68	49.53
2048	23.20	88.26
4096	23.19	176.62
8192	23.83	343.80
16384	26.16	626.40
32768	31.25	1048.64
65536	47.95	1366.81
131072	73.22	1790.16
262144	122.81	2134.50
524288	228.42	2295.28
1048576	430.26	2437.06

Table 3.4: SISCI DMA write performance at different transfer sizes.

Transfer size (bytes)	Latency (µS)	Bandwidth (MB/s)
4	0.14	27.98
8	0.13	60.24
16	0.06	271.43
32	0.06	516.15
64	0.07	958.85
128	0.08	1644.27
256	0.10	2569.12
512	0.19	2754.31
1024	0.38	2713.08
2048	0.74	2754.86
4096	1.47	2783.03
8192	2.95	2778.55
16384	5.89	2782.91
32768	11.74	2791.74
65536	23.54	2784.12
131072	46.86	2797.06
262144	91.81	2855.25
524288	183.23	2861.30
1048576	366.83	2858.51

Table 3.5: SISCI PIO write performance at different transfer sizes. This does not wait for the writes to be received on the other machine, and the results might be higher than the actual result.

A set of helper functions for performing optimized read or write requests to and from remote mapped segments are also provided. `SCIMemWrite` and `SCIMemWrite_dual` write from a local memory location to respectively one and two remote mapped segments. Similarly, `SCIMemCpy` offers both optimized read and write from remote mapped segments, with optional error checking also performed. `SCIMemCpy_dual` is similar to `SCIMemWrite_dual`, but also allows error checking. These functions can be used as alternatives to for example `malloc`.

3.4.5 Events

In addition to copying data between machines, the SISCI API also allows for some events to be observed. One of the events are interrupts, but there are also events associated with segments.

Events are handled with calls to functions that block the calling thread until an event occurs. For interrupts, this is a simple function that returns when the interrupt has been triggered, or if an error has occurred. No more information about the event is provided. For segments, which have multiple events, the wait function also returns the kind of event that occurred. A wait call, similarly to the one for interrupts and segments, is also available to wait for DMA transfers to complete. Segment events can be observed both for local and remote segments.

In addition to explicitly waiting for events, it is also possible to register callback functions that will be called when an event occurs. Callbacks are registered when the segment is created, and are implemented as threads that call the wait functions. When an event occurs, the callback function is called. The return value of the callback function can trigger two things. Either the thread waiting for events will wait for a new event, or the thread will exit.

When a callback is active, the wait functions cannot be used, so other ways of synchronization must be implemented. As the callback functions are called from another thread, access to shared resources must also be synchronized.

Because of the overhead of waking up a thread, there is a substantial delay for interrupts and other events compared to writing bytes directly to mapped memory. A typical remote interrupt trigger takes from 10 to 30 µS, compared to the much smaller delay of less than 1 µS when writing to mapped remote memory. For applications with extreme demand for minimal delay, spinning while waiting for a remote value to be changed can give much lower latency.

3.4.6 Error handling

While the underlaying hardware guarantees that the communication is error free, other errors can still occur. This can be programmer mistakes, or faulty or disconnected cables. To gracefully handle such errors, all the functions in the SISCI API take a pointer to an error variable as an argument. This variable is used to indicate the success or failure of the call. After each call to an API function, this variable should be checked to confirm that no errors have occurred. For asynchronous calls, like DMA transfers, error checking can be performed by checking the status of the transfer queue.

3.5 Distribution

As discussed in section 3.2, we will distribute the panorama processing pipeline with distribution modules chained into the pipeline. In this section we discuss how we have implemented this.

3.5.1 Distribution architecture

While we have decided to implement the image transfers between machines as standard module, we also need some control over the execution of the pipeline. Running the pipeline across multiple machines complicates simple operations, like starting or stopping a recording. We should preferably be able to control the pipeline from a central location. To solve this, the individual machines, which together make up the pipeline, must communicate with each other. There are multiple ways to do this. For our pipeline, we have considered two different distribution architectures: client–server and peer–to–peer [52].

Client–server
With a client–server architecture, there is usually one server node. All other nodes connect to this, and the communication flow always goes through this server. This is a simple setup. As long as the server is available, the clients always know where to connect to. No protocol for discovering other nodes are needed. It is also possible to create a hierarchal layout where servers are clients to other servers.

Peer–to–peer
The peer–to–peer architecture is the opposite of the client–server architecture. There are no servers or clients. All nodes are equal, and connect to each other. This is more robust, as there is no single point of failure. A peer–to–peer architecture is, however, inherently more complex. To begin with, we need an algorithm to discover other peers. This can be as simple as a pre–defined list of nodes, but it can also be a distributed discovery protocol. Next, we need to decide if all nodes connect to all other nodes. While this is possible in smaller networks, it becomes more complicated as the network grows. If all nodes does not connect to all other nodes, we also need to make sure that the network does not split, so that all information can be distributed to all nodes.

As our system is very small, only a handful of nodes, we can use both the architectures discussed above. Because of our scale, and relatively simple requirements, it makes most sense to use a client–server approach. We have no need for the extra fault tolerance added by the peer–to–peer architecture. By placing the server on a central machine in the pipeline, we guarantee that if the pipeline works, the server works, too. If a client disconnects, we can handle that according to the importance of that client. Should the server machine stop, the pipeline is most likely also broken. In these situations, it would not help to have a peer–to–peer setup, as the processing would stop anyway. In addition, the communication we have between machines is not time critical, so the extra step through the server is not a problem.

3.5.2 Pipeline layout

With the ability to move each module around in the pipeline, we need to decide on how to best distribute the pipeline out across our machines. We have three machines available. The specifications of these are shown in table 3.6 above. The machine named Cake is the one used by the previous pipeline. It has a six–core CPU, with plenty of PCIe lanes and lots of memory. The two other machines are commodity machines, with a quad-core CPU, limited PCIe lanes, and less memory.

Machine name	Cream	Topping	Cake
CPU	i7-2600 [53]	i7-2600 [53]	i7-3930K [54]
Chipset	P67 [55]	P67 [55]	X79 [56]
RAM	8 GB	8 GB	32 GB
PCIe version	2.0	2.0	2.0
PCIe lanes	16	16	40
PCIe layout	8–8	8–8	16–8–8

Table 3.6: Hardware specifications of the machines at Alfheim Stadion

One of the most important things we have to consider when deciding how to distribute the pipeline, is the number of available PCIe lanes in each of the machines. This is important, because we require multiple PCIe cards in each machine. To begin with, we need a Dolphin PCIe adapter in each machine. The cards we use are the IXH610 [57] adapters. For maximum bandwidth, these require 8 PCIe lanes. For connections to the cameras, and also for internal communication between the machines, we use Gigabit ethernet. Here we use Intel I350-T4 [58] (4 ports) or I350-T2 [59] (2 ports) cards, depending on how many ports we need. These cards use 4 PCIe lanes. In addition to the communication requirements, most of the processing in the pipeline is performed on GPUs. We therefore need a powerful GPU in at least one of the machines, which will require 16 lanes.

When it comes to available slots, Cake has three 16-lane slots and two 1-lane slots. The two other machines have two 16-lane slots and two 1-lane slots. After inserting the Dolphin cards, only a single full-length slot is left. Since these machines only have 16 PCIe lanes available from the CPU, we have 8 lanes left, as the Dolphin cards use 8 lanes. This rules out running a powerful GPU in these machines. The CPUs in these machines do have an integrated GPU, but because of the chipset used, this is not available for use. We still need to install a discrete GPU, because the machines will not boot without a GPU. After installing the ethernet adapters, we only have single slot connectors available. We have used NVS295 [60] cards which have been cut to fit in single lane PCIe slots.

With Cake being the only machine with enough PCIe lanes and slots to fit a powerful GPU, it must perform most of the processing in the pipeline. While it might be possible to run the entire pipeline on this single machine, this is likely to cause problems and intermittent delays in the pipeline. We have therefore chosen to move the camera capture onto the two other machines. We can only fit a single four port ethernet card into each of these machines, so we need both machines for capturing.

Possibly, we could also move the Bayer–to–YUV conversion to the Alienware machines and run it on the CPU. This will take more time than running it on a GPU. Additionally, it might affect the image capture. As we have enough resources to run it along with the rest of the pipeline on the GPU in Cake, we have decided to keep it there. Another benefit of keeping the Bayer–to–YUV conversion on Cake is the reduced bandwidth requirement. As Bayer is one quarter the size of YUV444, the delay of transferring data between the machines is significantly reduced.

Another processing step, that could be moved to free resources on Cake, is the H.264 encoding. As the encoding resources required vary greatly depending on movement in the image,

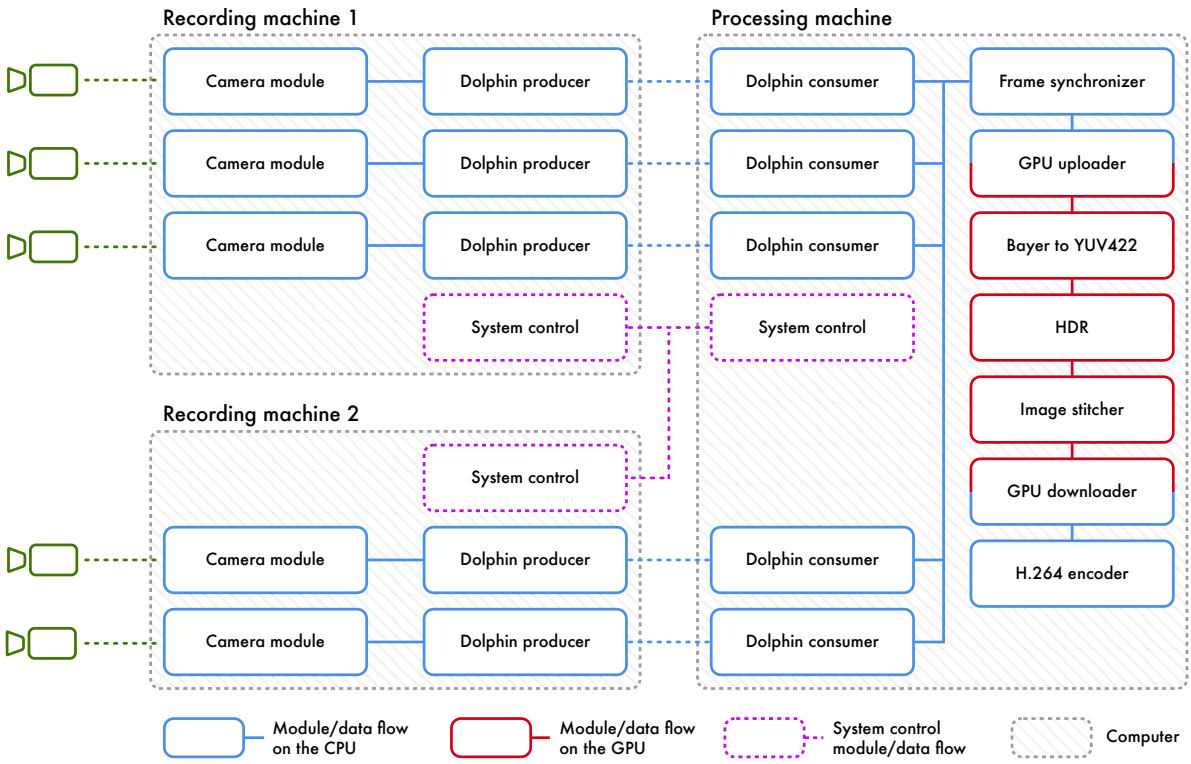


Figure 3.4: The initial distributed pipeline, with data flow between machines and modules.

the load from this step can sometimes spike for longer periods. Spikes in encoding times are common when there is more movement in the image than normally, like when snow is falling. Since we do not have extra machines available with enough resources to perform this step, it is not part of the current pipeline.

This leaves us with the pipeline setup shown in figure 3.4 above.

3.5.3 Image data transfers

The modules for transferring image data should behave like normal modules, and the data should be transferred with as little delay as possible. They should also be reusable, so multiple modules can be used at the same time. On the recording machines, the data transfer module will get frames from the camera capture module. This module is implemented as a normal fetch module, so the transfer module will be required to fetch frames from it. On the receiving side, the next module is the frame synchronizer, which is implemented both as a push and a fetch module. It requires frames from the individual cameras to be pushed to it, and the next module can then fetch the synchronized frame sets. Therefore, the receiving module needs to push frames to the frame synchronizer as they arrive.

When starting out with the modules for transferring data between the machines, we first took a look at what the SISCI API offers. It is important to create a solution that can scale beyond the current scale. We also consider it important that the modules could be used to transfer anything, not just frames. To ensure this, the modules will just transfer data of a predefined size from one

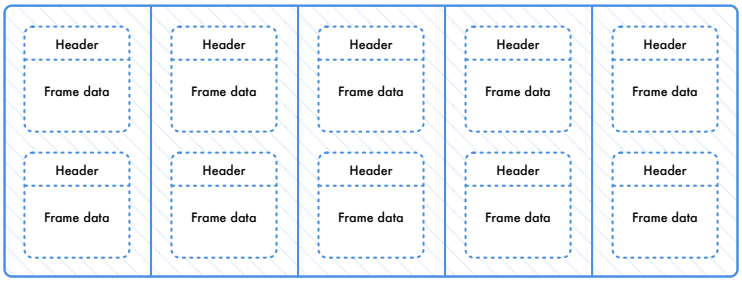


Figure 3.5: Transfer module memory layout with five cameras.

machine onto the next and then hand the data off to the next module. To make this work in a good way, we allocate memory for the header information and the frame data as one continuous block. The memory layout we ended up with, can be seen in figure 3.5 above.

This memory layout has several advantages over keeping the frame data and header information separate. First of all, it allows the transfer to be performed in a single operation. This simplifies the transfer and limits the overhead. As long as the transfer is performed in a single operation, the modules handling the transfer do not need to care about what is transferred.

To streamline the transferral and to keep the overhead to an absolute minimum, we additionally implemented the modules with two buffers per camera. This means that one buffer can be filled with data while the data in the other buffer is being transferred.

Having figured out how the data is going to be laid out in memory, it is time to look at the APIs offered for the transferring. Both remote mapped memory, accessed through programmed I/O (PIO), and DMA transfers would fill our needs. As we can see from tables 3.4 and 3.5 on page 29, PIO accessed memory starts with much lower latency at small transfer sizes, but is quickly surpassed by DMA requests as the size of the transfer increases.

High speed and low latency transfers are not the only factors that are important. As the tables show, both methods can achieve high enough bandwidth to support our needs. When transferring images in Bayer format, we need 525.28 MB/s of bandwidth for five cameras at 50 fps. Each frame is 2.10 MB. Therefore, we can use large enough transfers so that overhead will not be a problem.

When using PIO, an entire CPU thread is locked up while performing the transfer. While this is great for simplicity, it is not very efficient as the CPU could be spent doing other work while the transfer is being carried out. We therefore chose to use DMA for the data transfers.

With the data transfer taken care of, we had to look into synchronizing the buffer accesses on each machine. When a DMA transfer is taking place, there is no way for the machine receiving the data to know that it is happening. This means that a separate form of communication has to be used.

There are multiple solutions available here. We could use socket communication to send messages back and forth, but that would incur a great deal of delay. Another alternative is to use the events for segments in the SISCI API. This does, however, require the sender to either connect to or disconnect from the segment whenever a transfer has been completed. Using remote interrupts, we can signal the receiver directly when a transfer has occurred. While this is great for delays, we need another way of telling the receiver what buffer the frame was written to,

```
// Get the first frame
struct frame *buf = in->getResource((struct frame *)buffers[i]->buffer);

for (;;) {
    // Get the current buffer
    Buffer *b = buffers[i];

    // Wait for an interrupt from the receiver. When we get this interrupt,
    // we know that the buffer is ready to be written to.
    SCIWaitForInterrupt(b->ready_intr, 1000, NO_FLAGS, &error);

    // Start the DMA transfer to the remote data segment
    SCIStartDmaTransfer(b->dma_queue, b->local_seg, b->seg, 0, buffer_size, 0, NULL, NULL,
            NO_FLAGS, &error);

    // Wait until the last started DMA request has finished
    SCIWaitForDMAQueue(b->dma_queue, 1000, NO_FLAGS, &error);

    // Trigger the interrupt on the remote end
    SCITriggerInterrupt(b->intr, NO_FLAGS, &error);

    // Get next frame buffer. This should preferrably be no line 15, but because that
    // caused intermittent problems, it has been moved down.
    buf = in->getResource((struct frame *)buffers[i++ % 2]->buffer);
}
```

Figure 3.6: A simplified version of the code used to transfer the frame data between machines. Please note that all error checking and some other code has been removed to make it easier to read.

as no data can be sent along with the interrupt. We have chosen to solve this by using multiple interrupts, one for each buffer.

One important thing, however, is still missing. While the receiver of the data is notified whenever new data is available, the sender does not know when the buffer is ready to be used again. We have chosen to use interrupts here too, as it requires minimal code and is easy to use.

One issue with this setup is that it has dependencies in both ways. The sender is connected to the receiver, and the receiver is connected to the sender. This requires the shutdown to be performed in sets, and in the correct order, to avoid problems. In our prototype, we have not implemented this, since this does not cause any serious issues. It only generates a warning every time the pipeline is stopped. Improved modules for data transfers will be implemented and discussed in section 3.6. These solve this issue.

3.5.4 Control communication

To control the execution of the panorama processing pipeline, we rely on a message passing between the machines. Messages we need to send include the initial setup configuration, commands to start or stop recording, and exposure settings. As the messages' types and sizes varies, this communication is not as good a fit for the SISCI API as the image data transfers. While certainly possible, it would require extra effort to handle buffers and message parsing. This kind of communication is, however, a perfect fit for stream based communication with TCP sockets. Here buffers are handled for us, and all we need to implement, is our own message format, so we easily can read the messages as they arrive.

By using the Berkley API for doing socket communication, we can choose to communicate via the PCIe connection, using SuperSockets. This offers flexibility while allowing us to use familiar APIs.

Messages sent between machines follow the protocol shown in figure 3.7 on the following page. All messages have a header that contains the message type and the size of the message. The size field is used to check if the entire message has been received, and the type field is used to determine what kind of message that was received. With only these two values in the header, it can happen that invalid data falsely get recognized as valid data. To avoid this, a magic number[2] or a checksum could be incorporated into the header. This is not implemented in the current prototype.

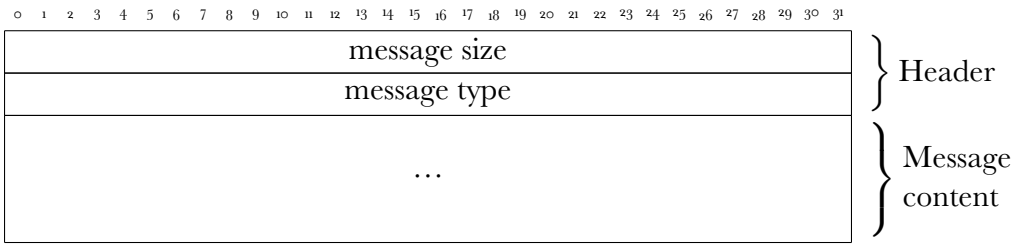


Figure 3.7: This shows the layout of the message protocol used to communicate between machines in the cluster.

[2]A magic number is one or more known numbers, used to validate the data received.

To simplify message handling, we have created a central communication handler. It takes care of receiving and sending messages. For the clients, this handles connecting to the server. On the server, it handles setting up a socket for receiving connections. Once a connection has been established, the server has a function to broadcast data to all receivers. While not true broadcasting, it sends the same message to all connected clients. On the client side, a function is available for sending messages to the server.

The different messages are used by different modules, and some are used by multiple modules. To simplify managing this, the connection helpers also handle the message receiving. This is done with a callback based interface. A callback function can be registered for a certain message type. The handler parses the header of all incoming messages, confirms that the entire message has been received, and then looks for any callbacks matching the message type. For all matching callbacks, the registered callback function is called, with the received message and an optional pointer provided when registering the callback, as arguments. This system simplifies handling of received messages, and also allow message types to be added or removed without changing the connection handlers. Additionally, by abstracting away the communication code, it is easy to change the communication protocol, without requiring changes to other code than the client and server helpers.

3.5.5 Summary

We have now shown how we can distribute the work load of the panorama processing pipeline: With modules that easily can be moved around, modules for sending data between machines and overall execution control across machines.

In the next section, we will show how we can improve the distribution.

3.6 Optimizing data flow

With the distributed pipeline up and running, we focused on improving the distribution setup, including optimization to the data flow. The original distribution code was written to transfer any data of a predefined size. This required the frame and meta data to be allocated together in the same buffer. While this works for frames in CPU memory, it does not work for frames in GPU memory. The frame data must be in GPU memory, and the frame meta data must be in CPU memory.

We also recognized that there was little or no need for transferring anything except for frame data using DMA requests. Compared to the size of the frame data, the meta information is tiny. The control communication between machines is also small. Based on this, we decided to improve the distribution code by making it data aware in the sense that it is specialized to transfer frames and meta data. This allows optimizations that would otherwise be impossible or hard to implement. It does, however, limit the flexibility.

3.6.1 Memory handling

The first step we took to improve the distribution code, was to separate the frame data from the meta data. This allows frames in CPU and GPU memory to use the same frame object, with just a pointer to the actual frame data. This also allows future changes, where the frame can be

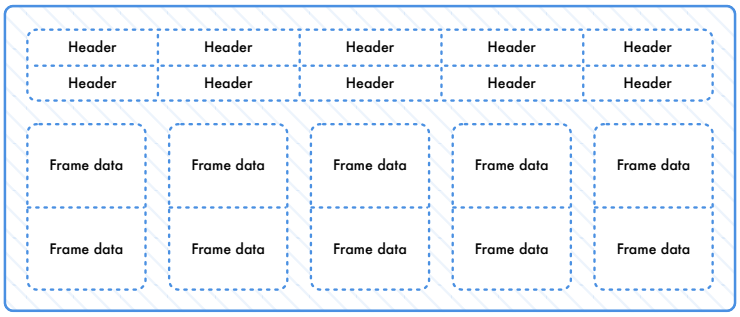


Figure 3.8: New transfer module memory layout, with 5 cameras

```
1  extern "C" struct frame
2  {
3      struct header   hdr;
4
5      // Union of different pointer types
6      union {
7          // CUDA runtime device pointer and normal CPU pointer
8          void       *ptr;
9
10         // CUDA driver API device pointer
11         CUdeviceptr cuPtr;
12
13         // CUDA driver API array pointer
14         CUarray     cuArrPtr;
15
16         // CUDA runtime API array pointer
17         cudaArray  *cuArr;
18     };
19 };
```

Figure 3.9: `C`-struct for storing frame meta data

located anywhere. The new memory layout can be seen in figure 3.8 above. Compared to the old setup, as seen in figure 3.5 on page 34, we now use six segments with five cameras, compared to ten before.

To separate the frame meta data from the image data, we use the `C`-struct shown in figure 3.9 on the next page. This contains the header defined in figure 3.2 on page 20. We have added a `union` of multiple pointer types. A `union` is a collection of multiple value types, where only one value is used at a time. The size of this field is equal to the size of the largest type included.

The `union` allows the same struct to be used, independently of where the image data is stored and regardless of what kind of pointer is needed. For example, frames in GPU memory are accessed with a different kind of pointer than frame in CPU memory. Using the information stored in the flags field in the header, we can check the type, and based on this, use the correct field for access. This eliminates the need to cast the pointer when using different pointer types.

3.6.2 Improved distribution modules

To be able to use the new memory layout, the distribution modules must to be updated. In addition to changing the modules to the new memory layout, we have also made them state aware. As the original distribution code was entirely based on interrupts and callbacks, it is almost stateless. While this simplifies the code, it also makes it hard to optimize. No information about the state of the frame buffers are stored. This means that the distribution modules could easily go out of sync if for example an interrupt was lost.

This turned out to be an issue that sometimes happened when running the pipeline. To improve this, we now store state in the transferral modules. The dependence upon interrupts have been almost completely removed. This does, however, have some challenges, as synchronization between the various machines must be handled in another way.

Circular buffers

There are multiple buffers for each camera, and each buffer is reused. This is a typical situation where a circular buffer is a good fit. A circular buffer is based on a list or an array, and works as a first–in–first–out list (FIFO) of constant size. There are usually two pointers associated with a circular buffer: one read pointer and one write pointer. The read pointer indicates the next element to be read from the buffer. The write pointer indicates the position where the next element should be inserted. As new elements are inserted, the write pointer is increased until it has caught up with the read pointer. The buffer is then considered to be full.

One problem with circular buffers is that it can be impossible to distinguish between a full and an empty buffer. When the buffer is initialized, both the read and write pointers usually point to the first buffer, indicating that the buffer is empty. If the buffer is then filled before any elements are removed, we end up in a problematic situation. The read pointer is still pointing at the first element, as that is the next one that should be read. The write pointer is also pointing to the first element, as that is where the next element should be inserted, as soon as the first element has been read. This is identical to the state the buffer was in when it was initialized.

There are multiple solutions to this problem. For a comprehensive list of possible solutions, see [61]. In our implementation, we have chosen to solve this problem by always leaving one empty slot in our buffer. This is not memory efficient, but with the new memory layout, we have enough free memory to use this solution. It is also the simplest solution to implement.

To further ease the situation, we have a single reader and a single writer. When implemented correctly, circular buffers with a single reader and a single writer can be used without locks. This is good for running across multiple machines, where synchronization can greatly increase complexity. While synchronization is possible via the SISCI API, it can add extra delays and will increase complexity. We want to avoid using synchronization steps where possible.

In circular buffers only the reader is allowed to update the read pointer, and only the writer is allowed to update the write pointer. This allows us to make certain assumptions. With only one reader and one writer, race conditions will never occur when inserting or reading from the buffer. The reader and the writer also operates in different ends of the buffer. If the writer completes the insertion into the buffer before updating the write pointer, the reader will see the

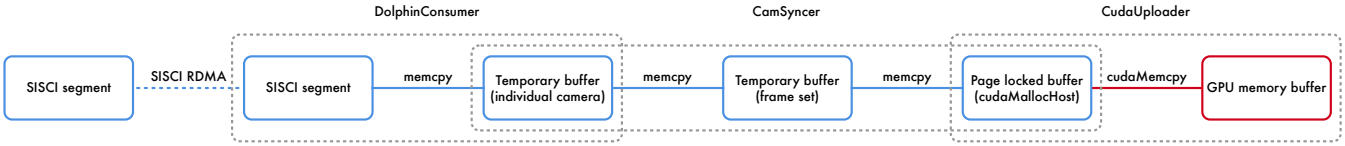


Figure 3.10: Memory copies being performed, from the frame is sent, using RDMA from the recording machine, until it is available for processing in GPU memory.

slot being inserted into as empty, until the insertion has been completed. Similarly, the reader must complete reads before updating the read pointer. This ensures that no partial data will ever be read.

The circular buffers we use in the pipeline, are implemented to keep all state on the receiving end. Since we operate across multiple machines, we cannot use pointers to indicate the read and write positions, and instead use indexes. The indexes are stored in a SISCI segment, which is mapped into the address space of both the sender and the receiver. These indexes actually indicate indexes for two parallel buffers, one for the headers and one for the actual frame data. This separation allows the frame and headers to be separated into different SISCI segments. Both the sender and the receiver can use the buffer indexes like normal memory, and operate as if it is a normal circular buffer, with the exception that the frame data is transferred with a DMA request. The buffer for the headers is mapped into the address space, just like the indexes. The headers are transferred using normal `memcpy` calls. This is because the headers are small in size, approximately 32 bytes. We can see from table 3.4 and 3.5 at page 29 that the overhead of a DMA request is large for small transfers like these.

Notifying the receiver

Unlike the interrupt based modules, this new implementation does not immediately notify the receiver when new data is available. This can be solved in two ways, either the receiver can constantly check the write pointer for changes. This guarantees minimal delay from a frame was written, until the receiver is aware of having received the frame. It does, however, also require extra resources to constantly check the write pointer, especially when frames usually only arrive 50 times a second. Another solution is to continue to use interrupts, but now only for notifying the receiver about a new frame, not maintaining state. Compared to constantly checking the write pointer, this alternative uses much less resources, for a small latency penalty. When constantly checking the write pointer, delays below 1 µs can be achieved, while interrupts typically take tens of microseconds. With our current work load, the resources saved by using interrupts are more valuable than the delay caused. So we have chosen the interrupt based model.

Minimizing memory copies

Next, we focused on improving memory handling. By looking at the data flow in figure 3.10 above, we can see that the data is copied at multiple places. This incurs extra delays and wastes resources. By limiting memory copies, we can reduce the pipeline latency and free up resources for other use.

On the processing machine, there are multiple places where optimizations would be useful. After the frames have been received by the processing machine, they are immediately passed to the frame synchronizer. There they are copied from the SISCI segment to a new CPU buffer. This is done to ensure that the SISCI segments are free for writing new frames as quickly as possible. When a complete set of frames have been received, another copy takes place. The individual frames of a set are copied from their buffers into a common buffer for the frame set. When `getFrame` is called on the frame synchronizer module, the frame set is copied from the common buffer to the buffer provided by the caller.

We now have a complete frame set, but no processing is performed on the CPU. The first thing that happens is that this frame set is copied from CPU memory to GPU memory. This is performed by the CUDA uploader module, which copies from a page locked CPU buffer, allocated through the CUDA library to the GPU.

Our first optimization step here is to minimize the number of memory copies performed before the upload to GPU memory. There are a couple of requirements that must be fulfilled. Firstly, images from the individual cameras must me synchronized and returned as a single set to the next module. In addition, we must keep the previous frame set around, so we can return it if a new set do not arrive in time. This leaves us with two options: either to keep the individual frames in their buffers until the next set has arrived, or store the frame set in a separate buffer.

We have chosen to keep the frame set stored in a separate buffer, as it reduces the number of buffers that must be available for other machines. It also simplifies the buffer handling code, as only unused frames are present in the individual camera buffers. We can use the same buffer for all calls to `getFrame`, as we can update it only when a new frame set is available. This way the old frame set is still present in the buffer as long as no new set has arrived. We then only need to update the header before returning it as a dropped frame.

This requires some changes to the module setup. As we no longer want to copy the frames out of the SISCI segments before they are synchronized, the frame synchronizer must to be changed. It needs to be aware of how the data is transmitted and notify the sender when it has removed a frame from a buffer. The sender must also be changed, as it no longer can rely on the transmitted frames to be quickly copied out of the buffers. To keep transmission flowing smoothly, it would also be desirable to have more than two buffers available.

The current transmission modules also make inefficient use of the addressable memory. This is limited by the PCIe design. A maximum of 22 segments can be made available for connections, and each with a limited size. Depending on the configuration, the size of these segments can be up to 32MB with the current implementation from Dolphin. One segment is always in use by the underlaying driver, so that leaves 21 segments available for use.[3]

With the first implementation, two segments were used for each camera. With the current setup of five cameras, it results in ten segments used. This does not scale well, if we add more cameras. Each frame is also limited in size, which results in poor use of available resources. With the current resolution of 2040×1080 pixels, we never use more than $1/7$ th of the addressable memory[4].

[3]The number and size of segments are limited by the Base Address Registers (BAR) of the PCIe specification.

[4]In HDR mode two frames are combined, resulting in a frame size of 4.41 MB. With a maximum segment size of 32 MB, the available memory is 7.26 times the frame size.

To increase the utilization of available memory, we have changed the setup. From using one segment per frame buffer, we now have multiple frame buffers in each segment. When using two buffers per camera, this increases our utilization to between 1/4 and 1/3. When running in HDR mode, we can use up to seven frame buffers in a single segment, and in non-HDR mode we can have 14 frame buffers in each segment. It is also possible to use a single segment for buffers from multiple cameras. We could, for example, use three buffers per camera, with two cameras in each segment. This way we could run up to 40 cameras. By reducing to two buffers per camera, we can use up to 60 cameras before running out of segments. [5]

When the pipeline is running as expected, there is usually no need for multiple frame buffers. The pipeline is not entirely stable though. To handle spikes in processing time, it is important to have buffers for multiple frames. This allows frames to be stored in memory while waiting to be processed. Without multiple buffers, we would risk spikes in processing times resulting in dropped frames. Because of this, we usually run the pipeline with at least three buffers per camera.

3.6.3 Integrated frame synchronization

To be able to better use these buffers, we decided to integrate the frame synchronization step into the data transfer module. This allows full control over the buffer usage. It also eliminates the need to copy single frames to temporary buffers. The move to use an integrated frame synchronizer also has other benefits, which will be discussed later in this chapter.

In the integrated frame synchronizer, we have implemented an iterative frame algorithm. The basic algorithm is shown in figure 3.11 on the next page. The algorithm works as following: it gets the first frame in the buffers of each camera. The timestamps of these frames are then compared to find the oldest and the newest frame. The difference between the oldest and newest frame is compared to a configurable threshold. If the difference is greater than the threshold, the oldest frame is removed, and the process repeats. When either an acceptable frame set have been found, or there are no more frames available from a camera, the loop is stopped. The number of frames in each camera's buffer is checked before any frames are removed, so that no frames will be removed until frames from all cameras have been received.

Configuring the threshold for maximum difference between frames in a set to a minimum is important, as the synchronizer can potentially return frames from different sets. This can happen if the threshold is too large. If the images are captured very close to each other, for example by using an external trigger signal, the threshold can usually be set to a very small value. However, if there is a large variance in the capture times of the frames, the threshold must be set larger. In these cases, the frame sets returned from the synchronizer, can be different, depending on the order in which the frames are received. Because frames are checked as they are received, a worse match might be chosen, if it is within the threshold, even if some of the frames actually belong to another frame set. The synchronizer is therefore best suited for usage with highly synchronized frames, and a minimal threshold.

[5] One segment is used for headers and other meta data, so we have 20 buffers available for frame data.

```
while (true) {
    // Loop through the cameras and find the oldest and newest frame
    for (int i = 0; i < cams.length; i++) {

        // Check that we have at least one frame available from the camera
        if (availableFrames(i) < 1) {
            return false;
        }

        // Get the oldest frame of camera i
        frame = getFrame(i);

        if (isOlder(frame)) {
            oldest = frame;
        }

        if (isNewer(frame)) {
            newest = frame;
        }
    }

    // Check if the difference between oldest and newest frame is small enough
    if (diff(oldest, newest) < threshold) {
        return true;
    } else {
        removeFrame(oldest);
    }
}
```

Figure 3.11: A simplified version of the code used to synchronize frames from the individual cameras into frame sets. *Please note that this is not the actual code used.*

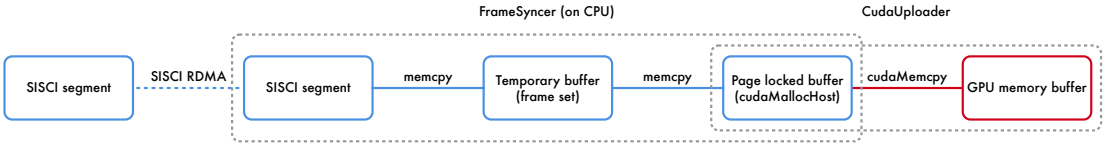


Figure 3.12: Memory copies being performed from the frame is sent, using RDMA from the recording machine, until it is available for processing in GPU memory.

For the first frame of a new recording, these steps are repeated until an acceptable set of frames have been found. This guarantees that we always start out with a perfectly matched set of frames. It does, however, mean that if one or more cameras are broken, no data will ever be recorded. This can be fixed by setting a deadline when frames from all cameras must be received, allowing it to fall back to running without the missing cameras.

After receiving the first set of frames, a deadline is saved one and a half frame interval in the future. This deadline is used in the next frame request. If no complete set of frames has been received before this deadline, the last set of frames is returned, with a flag signaling that the frame was dropped. This is done, because we always want to have a constant frame rate in our output files, and a single repeated frame is hard to notice.

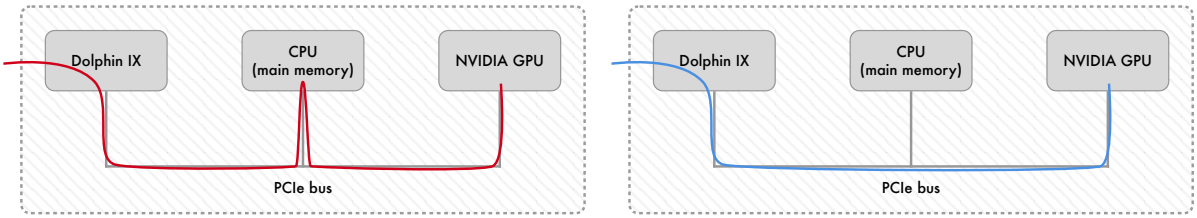
With the integrated frame synchronizer, we have removed one memory copy, and the data flow before processing now looks like in figure 3.12 above. There are still many memory copies being performed, so we want to further improve the memory handling.

3.6.4 GPUDirect RDMA

In addition to better utilization of available segments and memory, the integrated synchronizer also allows us to improve performance by moving the frame buffers from CPU memory to GPU memory. As the pipeline overview in figure 3.4 at page 33 shows, the frames are immediately uploaded to the GPU after being synchronized. Both the Dolphin card and the GPU is connected to the PCIe bus, and the PCIe bus allows peer–to–peer communication. This means that the cards can communicate directly with each other, without involving the CPU. In our pipeline, it would make sense to copy the image data directly from the Dolphin card to the GPU, without going through CPU memory. Figure 3.13 on the next page shows a visualization of the data flow, with and without GPUDirect.

Most of the processing in the pipeline is performed on GPUs. When transferring data between machines, the data currently has to be transferred between CPU memory in the two machines. This imposes extra latency, as the data must first be transferred to the machine and then transferred from CPU memory to GPU memory. Another drawback of this is that an extra step is required. The receiving machine must then be notified when data is available in CPU memory, as it must transfer the data from CPU memory to GPU memory.

Dolphin has already implemented support for connecting to peer–to–peer devices in their SISCI API [62]. To connect to a device, the IO address of the memory must be known. This is required as the IO addresses are used by the Dolphin card to read or write from the memory. The CUDA API hides these addresses from the programmer. Nvidia does, however, have APIs to



a) Without GPUDirect RDMA

b) With GPUDirect RDMA

Figure 3.13: (**a**) shows the data flow without using GPUDirect RDMA and (**b**) the data flow when GPUDirect is being used.

access the IO address. Nvidia calls this GPUDirect RDMA [4]. It allows access to IO addresses of memory on the GPUs, through a combination of users space and kernel space APIs. This API is only available on Tesla and Quadro products.

The API has changed from CUDA version 5.5 to 6.0. As we implemented this functionality before CUDA 6.0 was released, we will first introduce how it worked in version 5.5, and then outline the changes in 6.0. While the 5.5 way of doing this is deprecated, it does still work as of version 6.0 of CUDA.

The GPUDirect API

First of all, memory must be allocated by the application. This is done through the normal CUDA APIs like `cudaMemAlloc`. To access this memory through peer–to–peer, there are several steps that must be performed.

Next, a set of tokens must be acquired. This is done with the `cuPointerGetAttribute` function. By passing the `CUDA_POINTER_ATTRIBUTE_P2P_TOKENS` flag to this function, two tokens are returned. These two tokens together identify the virtual address space, and are used for authentication in the kernel space API. This is done to ensure that only the process that allocated the memory, can access it. When these tokens have been acquired, they must be passed from user space to a device driver in kernel space, for usage with the second part of the GPUDirect API.

In kernel space, the tokens, together with the virtual address, are used to access the actual IO address of the physical memory. This is done with a call to `nvidia_p2p_get_pages`. In addition to the tokens, virtual address, and size of the memory, this function takes several other arguments: A double pointer to a page table structure, which is set to point to a page table upon success. Next, a callback function, and optional argument to this callback function, must be provided. The virtual address of the memory to be pinned, should also be aligned to a 64 kilobyte boundary. This is because memory in the GPU's Base Address Register (BAR) is mapped in 64 kilobyte blocks. Calls to this function are expensive, as they require reconfiguration of the PCIe BAR tables of the GPU. Because of this, it should be called as rarely as possible.

The page table structure returned by this function upon success, is allocated by the Nvidia driver and should not be freed like normal memory. This memory is managed with additional parts of the API. When the application is finished using the pinned memory, the memory mapping and page table structure should be returned to the Nvidia driver through a call to

the `nvidia_p2p_put_pages` function. This takes care of removing the memory mapping, and frees the page table structure. After a call to this function, the mapped memory is no longer available for peer–to–peer communication. This call also requires the PCIe BAR tables to be reconfigured, and is thus also an expensive call.

In the event of the memory pointed to by the page table being freed, for example by the user through `cudaFree`, the callback function will be called before the memory is actually freed. This allows the kernel module to wait for interactions with the memory to complete, before cleaning up. `nvidia_p2p_free_page_table` should be called by the callback function, to free the page table structure. After the callback has completed, the memory is no longer available for peer–to–peer communication.

From version 5.5 to 6.0, Nvidia has deprecated the token-API. The tokens requested in user space, are no longer needed. The process that allocated the memory, can pin the memory for peer–to–peer usage without tokens. Instead of passing tokens between user space and kernel space, a new function has been introduced; `cuPointerSetAttribute`. This is a user space API that tells the driver that the memory, pointed to by a given virtual address, should be made available for peer–to–peer access. The kernel space API is unchanged, with the exception of the possibility to use 0–s instead of the tokens, when requesting the page tables.

Implementing GPUDirect support

As parts of the GPUDirect API are in kernel space, a kernel module is required. There are two possibilities for implementing this. The token and page table handling can be implemented into the main device driver, or it can be implemented as a separate kernel module dedicated to translating the virtual CUDA addresses. Both alternatives serve the same purpose.

Having the token handling and memory pinning integrated into the main device driver, simplifies the setup. Only one driver must be developed, managed, and supported. This does, however, require the driver to be compiled and linked with the Nvidia driver. This makes it less flexible, and it makes the driver dependent on a third party driver. It also requires the driver to be recompiled for new versions of Nvidia's drivers, and it must be recompiled, if the Nvidia driver is no longer going to be used.

Developing a separate kernel module to handle the interaction with the Nvidia driver, removes the dependencies. Instead of being in the main driver, they are now in a separate module. This adds complexity, because two drivers must be maintained.

In cooperation with developers at Dolphin, we have implemented support for GPUDirect into the main SISCI driver.

Using GPUDirect in the pipeline

As outlined, we use GPUDirect in the pipeline to move the frame buffers from CPU memory to GPU memory, and then write directly to that memory with DMA requests. This is beneficial, as it removes the intermediate steps in CPU memory. As almost all the processing in the pipeline is performed on GPUs, there is no need to have the frame data in CPU memory at all, until the processing is complete and the image is ready for encoding.

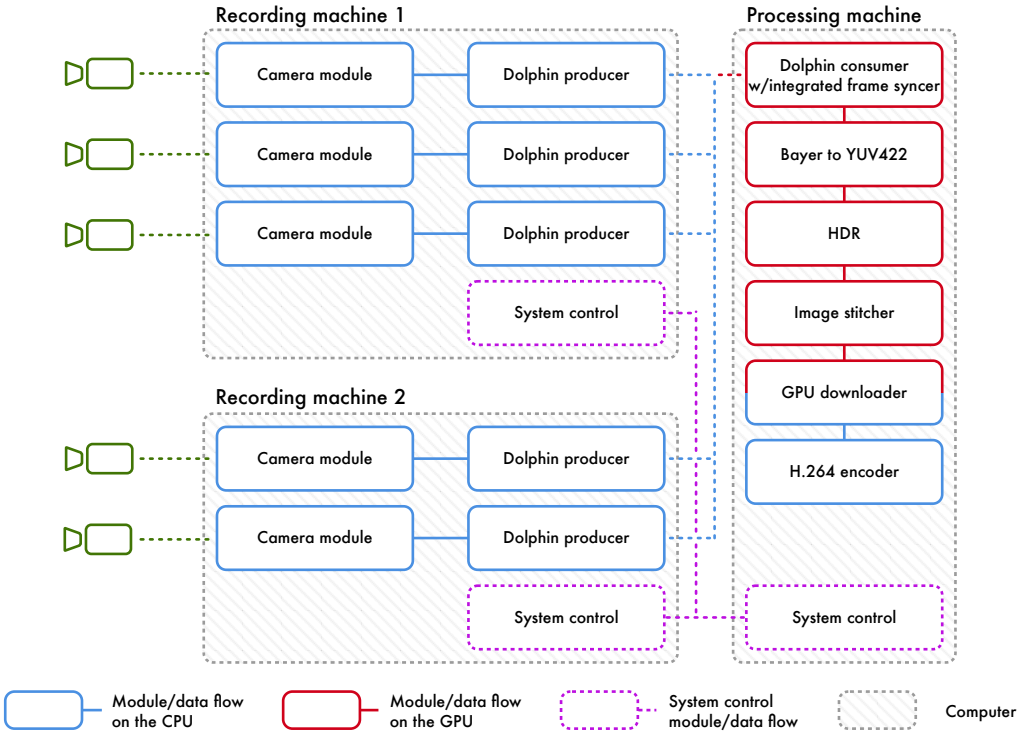


Figure 3.14: The improved distributed pipeline layout

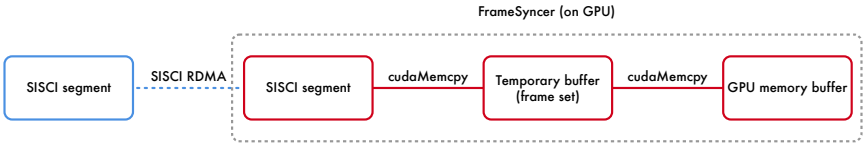


Figure 3.15: Memory copies being performed from the frame is sent using RDMA from the recording machine, until it is available for processing in GPU memory.

What we do need to be accessible from the CPU, are the frame meta data headers. As this information is only used by the application on CPU, there is no need to have this information in GPU memory.

With GPUDirect implemented, the pipeline looks like figure 3.14 on the next page. The memory flow between transferral and processing is shown in figure 3.15 on the next page. We have removed the entire CUDA uploader module, and limited the number of memory copies to a minimum. Compared to where we started, we have removed two memory copies.

In chapter 4 we evaluate the improvements we have implemented here.

3.7 Alternative distribution setups

With the dynamic distribution setup, we have the ability to move modules freely around. This allows a large number of possible distribution setups. With new machines purchased for our prototype setups, the possibilities are even greater than before. All the new machines have CPUs

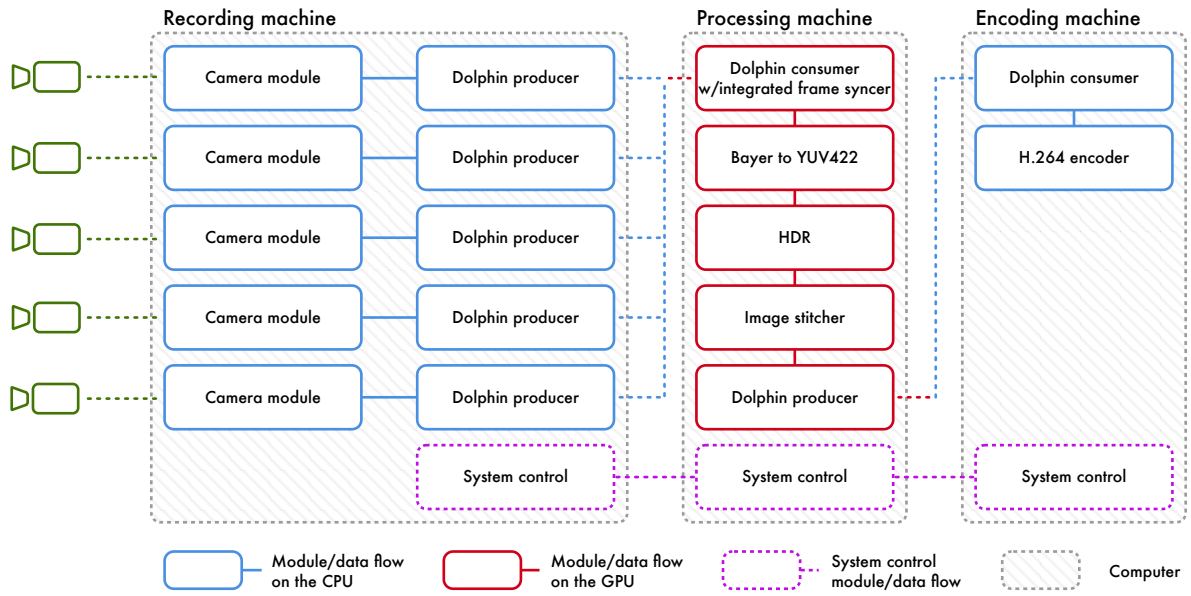


Figure 3.16: Pipeline with dedicated encoding machine

with 40 PCIe lanes. This allows new setups not possible with the machines in use at Alfheim Stadion now. We will discuss a couple of possibilities here.

All setups hare are shown with GPUDirect, but they are also possible without GPUDirect. This will add delay, as a memory copy from CPU to GPU is required. The setups shown here also use a single recording machine with 5 cameras. This is not required. Two or three recording machines, like the setup used at Alfheim Stadion, is also possible.

3.7.1 Separate encoding machine

One possibility is to move the encoding module to a separate machine, dedicated to encoding. This is shown in figure 3.16 on the next page. This enables the use of a less powerful machine for the processing steps, as all modules left on the processing machine run on the GPU. This leaves the CPU in the processing machine largely unused, reducing the risk of issues with thread scheduling. With no other modules running on the encoding machine, all CPU resources can be dedicated to the H.264 encoder.

This setup also benefits from splitting the pipeline in the places where the frames are at their most compact. Between the camera modules and the frame synchronizer, the frames are copied in Bayer format. Between the image stitcher and the H.264 encoder, the format is YUV422. This requires 4406 Mbps between the recoding machine and the processing machine, and 5505 Mbps between the processing machine and the encoding machine.

When using GPUDirect, the frames can also be copied directly from GPU memory on the processing machine to CPU memory on the encoding machine. This removes the need for a module to download from GPU to CPU memory.

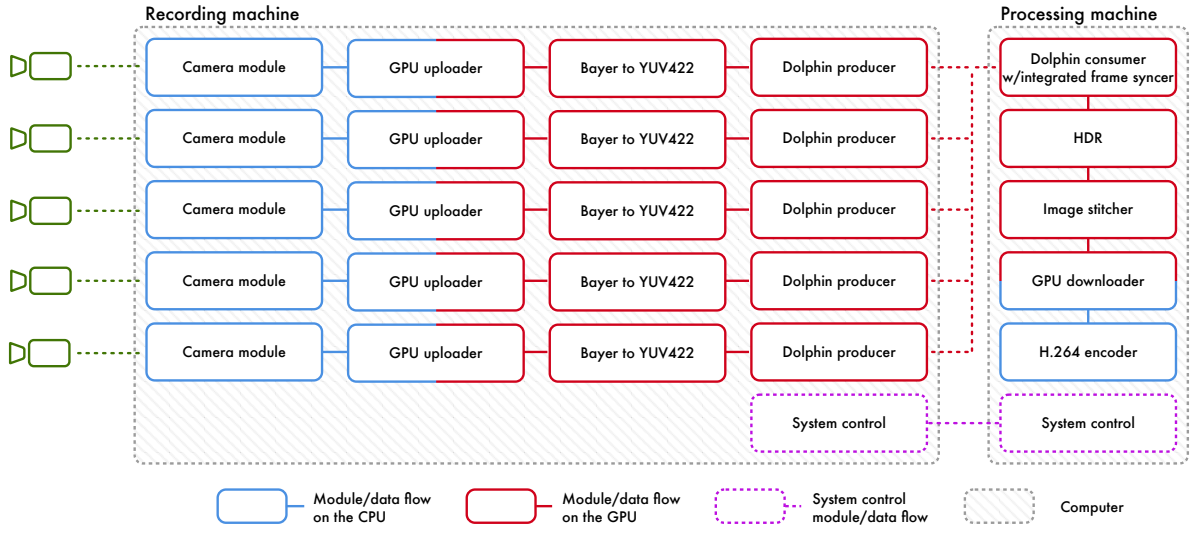


Figure 3.17: Pipeline with be–bayering on the recording machine

3.7.2 De–bayering on the recoding machine

Another possibility is to move the de–bayering module to the recording machine. We then perform the conversion from Bayer to YUV444 format before transferring the frames to the processing machine. This allows us to use GPU resources in the recording machine, and reduces the load on the GPU in the processing machine. This setup is shown in figure 3.17 on the next page.

As the de–bayering is performed on GPUs, we must add a module to copy from CPU memory to GPU memory on the recording machine. This will add some extra delay to the pipeline. The output of the de–bayering module is YUV444. Compared to other setups, where frames are transferred in Bayer format between the machines, this increases the size of the transferred frames from 1 byte per pixel (bpp) to 4 bpp [6]. This will likely result in an almost quadrupled transfer time, as we are already using large enough transfers to reach close to maximum bandwidth.

3.7.3 Two processing machines

A third possibility is to split the pipeline in the middle, between the HDR module and the image stitcher. This will separate the most resource demanding modules of the pipeline. Figure 3.18 above shows this setup.

This requires both processing machines to have powerful GPUs. When using GPUDirect, this setup will require two expensive GPUs. Because of this, setting up the pipeline like this at the current scale is not sensible. Without GPUDirect, this setup is likely to add significant latency. It also has the same drawback as moving the de–bayering. Between the HDR module and the image stitcher module, data is transferred in YUV444 format.

[6] YUV444 is normally 3 bpp, but in our pipeline it is padded with an additional byte for alignment to 32 bit boundaries.

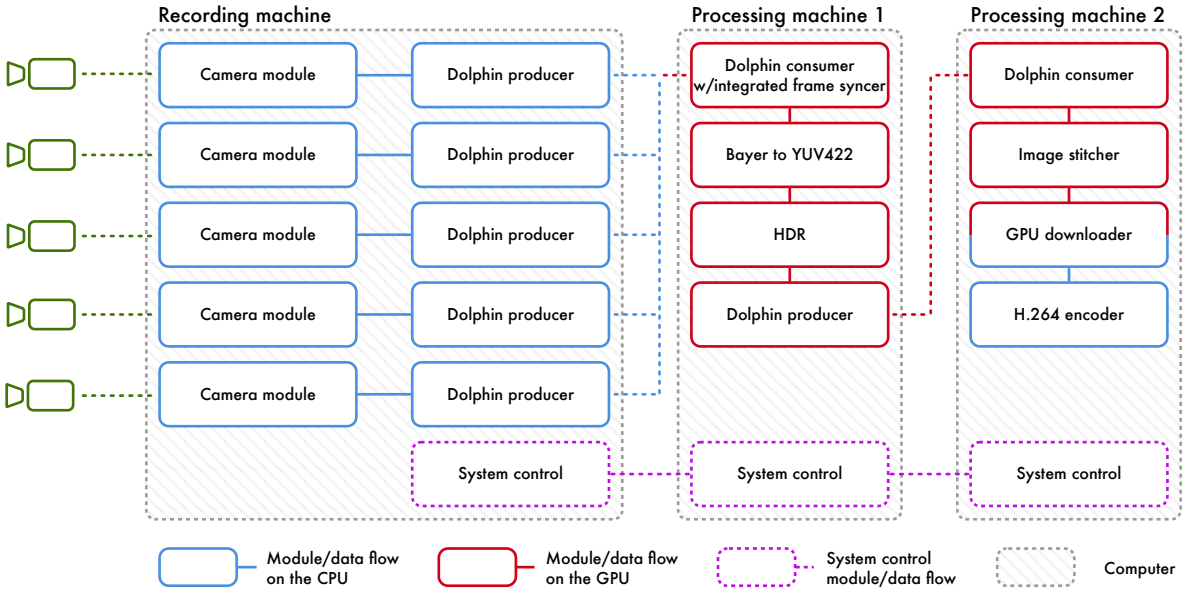


Figure 3.18: Pipeline with two processing machines

Compared to a setup with a separate encoding machine, this setup is likely to be slower. As long as a single GPU is able to perform all processing steps in our pipeline, splitting the processing across two machines will result in very small gains.

3.8 Future improvements

While we have optimized large parts of the distribution, there is still room for further improvements. Right now, all frames are always transferred from the recording machines to the processing machine, and then copied from individual frame buffers into frame set buffers. This is an extra memory copy that could potentially have been removed, if the transfer modules where smarter. If the synchronization algorithm had been distributed, the frames could be transferred directly into frame sets, removing the need to copy frames more than once on the processing machine.

In addition, a distributed synchronization could skip the transferral of dropped frames, and thus not waste bandwidth transferring frames that will be thrown away anyway. However, as the average drop rate of the pipeline is very small, less than 1 %, this will not make a big difference.

For further development of the pipeline, more advanced ways of distributing the processing would probably be required. The encoding step is a good example of this. We use the `x264` library, which is not written for running in real–time. Because of this, the time it takes to encode each frame is very inconsistent. Since we encode the video in segments, we could possibly split the encoding over several machines, in time–slices matching the segment size. This would require a distribution module that could switch between multiple receivers.

Additionally, the Dolphin IX interconnect offers native broadcast. This could be used for splitting the pipeline where multiple modules operate on the same input. This could be useful for performing steps like background subtraction in parallel to the encoding step.

A single transferral module that can receive frames from multiple cameras, is also a possibility to improve performance and minimize work load. The transferral modules used now, have one thread per camera. Using a single module, we can reduce this to one or two threads. With camera modules that can buffer frames, we can increase the delay between `getFrame` calls. This will reduce the time spent while waiting for a frame to receive. By using two threads, we can have one thread perform transfers, while the other thread requests frames from the camera modules. In combination with camera modules that can handle multiple cameras, this will reduce the number of threads even more.

3.9 Summary

In this chapter, we have discussed the problems we face in upgrading the panorama processing pipeline and possible solutions to these. We have introduced the module interface used for interaction between modules. This includes standard interfaces for getting frames from another module, and a common `C`–struct to represent frames with meta data.

Furthermore, we compared interconnect solutions, and how they can be used to distribute the processing in our pipeline. We have created modules for sending frames between machines, and set up handlers to control the pipeline when it is running distributed. We also optimized the initial distribution setup to remove delays and reduce resource use.

In the next chapter, we benchmark and evaluate the performance of the proposed distribution setup.

Chapter 4

Evaluation and results

In chapter 3, we have shown how we have modularized and distributed the Bagadus panorama pipeline. Like the previous version of the pipeline, the distributed pipeline also runs in real–time. This requires high and consistent performance. To ensure that our implementation delivers, we must benchmark and evaluate its performance.

In this chapter, we will look at the performance of the distributed pipeline. We first introduce the hardware used for testing. This includes new machines, purchased for running the pipeline, and some older machines. Next, we introduce the tests we have run. We have used a set of synthetic benchmarks to test the performance of parts related to the distribution, like raw bandwidth and latency. In addition, we have timed the actual pipeline. This gives us an overview of how the pipeline actually performs. After running benchmarks, we evaluate the results and discuss how they affect our pipeline.

4.1 Test setup

To evaluate the new pipeline setup, we have run several benchmarks. We have used some benchmarks to evaluate how the performance is affected by machine setup and different chipsets and PCIe switches. Using the information from these benchmarks, we can optimize our machine configurations to eliminate bottlenecks. We have also benchmarked the processing pipeline.

4.1.1 Hardware

Machines

All benchmarks have been run on machines in our lab. Most of the tests, and all of the pipeline tests, have been run on new machines. These are machines purchased for our prototype setup at Alfheim Stadion and for a similar setup at our lab. We will install the new machines at Alfheim Stadion in the autumn 2014.

Table 4.1 on the next page shows the hardware specifications of the new pipeline machines. The two machines, named Surf and Roundhouse, have been purchased as replacements for the existing recording machines. The motherboards in these machines are based on the X79 chipset. They also have two PLX8747 [63] PCIe switches to accommodate for extra extensions via PCIe. Each of the PCIe switches are connected to the CPU with a third generation 16-lane PCIe bus.

Machine name	Surf / Roundhouse	Meaningless
CPU	i7-4820K	i7-3930K
Cores/Threads	4/8	6/12
Chipset	X79	X79
Motherboard	Asus P9X79-E WS	Intel DX79SI
RAM	16 GB	32 GB
PCIe lanes	40	40
PCIe layout	16–16–8–16–16 / 16–8–8–8–16–8–8	16–8–8

Table 4.1: New pipeline machines

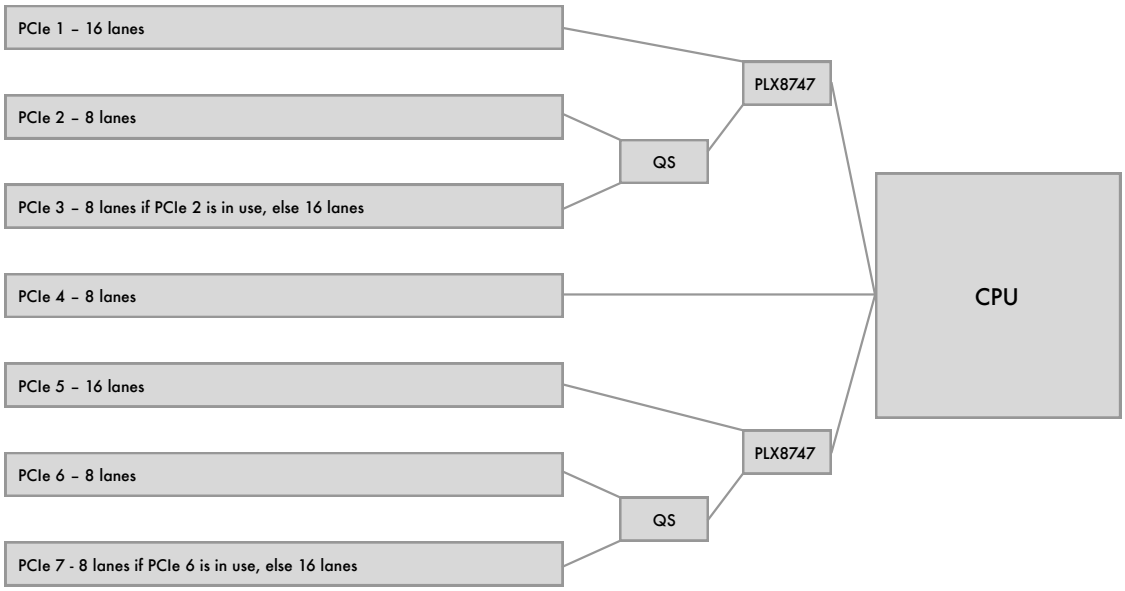


Figure 4.1: PCIe slots on the Asus P9X79-E WS motherboards. Original figure from [64].

Three PCIe slots are connected to each switch. The first one is always 16-lanes, the second is always 8-lanes, and the third is 16–lanes when the second slot is unused and 8–lanes when the second slot is in use. In addition, a single 8-lane PCIe slot connected directly to the CPU is also available. Figure 4.1 above shows a diagram of how the PCIe slots are connected. Because of the high number of PCIe slots, these machines are perfect for testing the performance of PCIe peer–to–peer communication in different configurations.

In addition to the recording machines, we are still using the original processing machine. The machine we have at our lab is called Meaningless. This is identically configured to the machine called Cake in Tromsø. This machine also has a X79 chipset, but without PCIe switches. This gives a maximum of 40 PCIe lanes. The motherboard in this machine has three full–length PCIe slots, and two single–lane slots.

For comparing performance between the current and previous generation chipsets from Intel, we have used an older machine with the X58 chipset. The specifications of this machine can be seen in table 4.2 above. We also tried to get GPUDirect to work on a machine with the newer

Machine name	Powerless	Singapore
CPU	i7-960	i7-4770
Chipset	X58	Z87
Motherboard	Asus P6T	Asus Gryphon Z87
RAM	6GB	8 GB
PCIe lanes	40	16
PCIe layout	16–16–4	16–4 / 8–8–4

Table 4.2: Additional machines used for benchmarking

generation Z87 chipset, the machine called Singapore in the table above, but we did not get it to working.

GPUs

Nvidia has only enabled GPUDirect access on their workstation and data center product lines, the Quadro and Tesla products. This limits the number of available GPUs we can test GPUDirect performance with. We have used two different Nvidia GPUs, shown in table 4.3.

	Quadro K2000	Tesla K20c
CUDA cores	384	2496
Memory	2 GB	5 GB
Memory bandiwdth	64 GB/s	208 GB/s

Table 4.3: GPUs used for benchmarking

The K2000 is a low–end workstation GPU, with 384 CUDA cores. It is capable of running our processing pipeline, but not at high frame rates. Because of the limited memory and number of CUDA cores, the processing takes significantly longer than on higher end products.

The K20c is a high–end data center GPU. It has 2496 CUDA cores and lots of memory. This card is capable of running our pipeline at all frame rates. It has been used for all tests of the pipeline shown in this chapter.

Both these GPUs are second generation PCIe devices. With up to 16 lanes, the theoretical bandwidth of these cards are 8 GB/s.

Dolphin IX cards

We have used Dolphin IXH610 cards. These are second generation PCIe cards, that use up to 8 lanes. The cards are based on PCIe standards, and operate as PCIe bridges. It is possible to use the cards as both transparent and non–transparent bridges. As we use the cards to communicate between machines, we use them in non– transparent mode.

Given that the cards are second generation PCIe devices, each lane delivers 250 MB/s of bandwidth. With 8 lanes, the Dolphin IX cards have a theoretical max bandwidth of 4 GB/s across the PCIe link.

4.1.2 Tests

To evaluate the performance of the distribution setup, we have used a combination of raw performance benchmarks and timing of the actual pipeline. This gives us a good picture of the performance we can expect, while also helping us understand whether we have introduced actual overhead in the pipeline by comparing benchmark results with actual pipeline performance.

DMA benchmark

To evaluate the performance of DMA requests, we have used a benchmark provided along with the Dolphin drivers, called `dma_bench`. This is a simple benchmark that runs a series DMA requests of the same size and logs the time required to complete all transfers in the series. By dividing the time required by the number of transfers, we get the average time taken to perform each request, including overhead. Since we know the size of each transfer, we can calculate the total bandwidth achieved.

The timing is done with the `RDTSC` instruction in x86. This instruction returns the number of ticks since and arbitrary points in time, usually when the CPU was powered on. The tick count before the tests was run, is then compared to the tick count after running the tests. We then know the number of ticks taken to run the tests. Since we also know the processor speed, we can calculate the time taken from the difference in tick counts.

The version of `dma_bench` provided by Dolphin does not support allocating and using memory on GPUs. We have therefore extended the tool to add an option allowing for memory to be allocated on a GPU. The memory allocation is performed using the `cuMemAlloc` function, and it is associated with a SISCI segment using the `SCIAttachPhysicalMemory`.

The DMA requests in this benchmark are performed using the normal SISCI API. So the results show actual application performance. This tool has been used to compare the bandwidth of different setups. First of all, we have compared bandwidth between using a segment on GPU accessed using GPUDirect versus a normal segment allocated in CPU memory. We have also used this tool to evaluate the bandwidth achieved for different setups, like different chipsets and with or without PCIe switches.

CUDA peer-to-peer

In addition to the Dolphin related benchmarks, we have also run a CUDA benchmark provided by Nvidia called `p2pBandwidthLatencyTest`. This is a test of memory bandwidth between CUDA devices. By comparing the results obtained from this test with the results from our Dolphin tests, we can see if these results are similar or not. This test uses `cudaMemcpyPeerAsync` to perform memory copies between devices.

Pipeline

To see how pipeline performance is affected by the different setups, we have also benchmarked the actual pipeline. This has been done with a slightly modified version, where the current timestamp is stored at different points throughout the pipeline.

	Input	Output
3 cameras	2040×1080×3×1 bpp = 6.30 MB	2848×1864×2 bpp = 10.13 MB
7 cameras	2040×1080×7×1 bpp = 14.71 MB	5920×1680×2bpp = 18.97 MB
HDR mode	2040×1080×6×1 bpp = 12.61 MB	2848×1864×2 bpp = 10.13 MB

Table 4.4: Size of panorama input and output for different setups.

As the pipeline runs across several machines, the timestamps used are from the real–time clock. This ensures that the timestamps can be compared across machines. It also means that the measurements can be affected by differences in each machines clock, and also by clock synchronization tools like `ntp`. To ensure minimal differences between the machines clocks, a manual synchronization was run before the tests. All timestamps are collected with the `clock_gettime` function. This returns a unix timestamp in seconds, plus the number of nanoseconds since that second. This function can use multiple sources. The accuracy of the timestamp, and cost of a call the function, varies based on the source [65].

Since the encoding step has varying latency and is performed synchronously, it can have great effects on the pipeline stability. When looking at the performance of the distributed pipeline, the encoding step is irrelevant. Some of the machines we have used for testing, are also unable to encode the panorama video in real–time, causing many dropped frames. We have therefore disabled the encoding step in all pipeline benchmarks.

We have used various configurations of the pipeline in our testing. In table 4.4 above we have listed the data input and output size for each configuration. In HDR mode, two images from each camera are grouped together. While the cameras are capturing frames at 50 fps, the effective frame rate is 25 fps. All HDR tests we have run, are with three cameras, unless otherwise stated.

4.2 Benchmark results

Each of the tests have been run in multiple scenarios, to compare setups and discover possible bottlenecks. We here shortly introduce each of the tests that have been run explain and why we have chosen to run these specific tests.

4.2.1 Synthetic benchmarks

Chipsets and PCIe switches

We knew from the GPUDirect documention [4] that the performance of peer–to–peer access with GPUDirect could be greatly affected by the chipset and PCIe layout. As described above, the Asus P9X79-E WS motherboard has PCIe switches, which according to the GPUDirect documentation should offer optimal performance. To investigate this, we have run a series of tests with the `dma_bench` test. First of all we have run the test with a Dolphin IX card and a Quadro K2000 GPU in different slot setups.

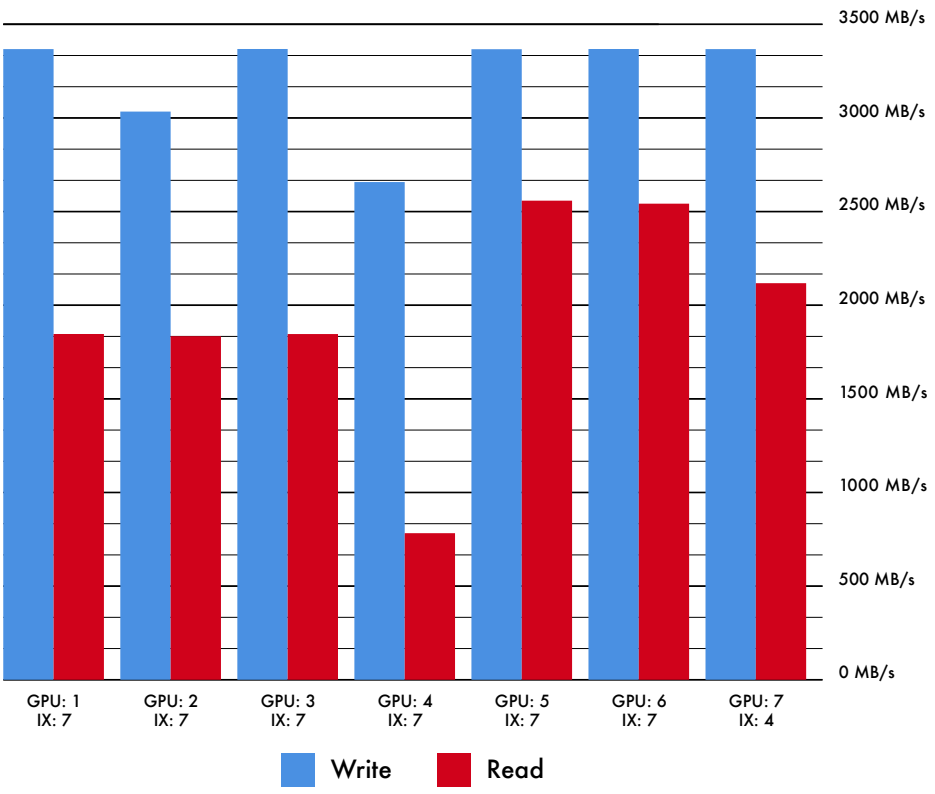


Figure 4.2: This chart shows differences in DMA bandwidth between different PCIe slots on an Asus P9X79-E WS motherboard. The blue bars are bandwidth achieved writing to GPU memory, and the red bars are bandwidth achieved reading from GPU memory. All numbers are from reading or writing 2 megabytes, which is comparable to the size of each of our frames in Bayer format.

The results are shown in figure 4.2 on the previous page. The x axis labels show the PCIe slot of the GPU and the Dolphin IX card. These slot numbers are described in figure 4.1 on page 54. All tests are run with no other PCIe cards installed. As we can see, using the PCIe switches definitely gives the best performance. It also appears that when going via the CPU, having the GPU in a 8– lane slot impacts write performance. For all other setups, when the GPU is in a 16–lane slot and when the GPU is in the same PCIe switch, the write speed seems to be consistent. Read performance is always slower than read, even when the slots are connected to the same PCIe switch. Read performance does not appear to be affected by the GPU being placed in an 8 or 16–lane slot, but it is affected by going through the chipset, as can be seen by the read bandwidth being lower both when the cards are placed in different PCIe switches and when the GPU is placed in the PCIe slot connected directly to the CPU.

Bare four and seven in figure 4.2 are showing the opposite setups of each other, where the GPU is placed in the seventh PCIe slot and the Dolphin IX card is placed in the fourth slot. It appears to be a problem with this slot and the GPUs, as read bandwidth is extremely low when the GPU is in slot 4, but much higher if the Dolphin IX card is placed there instead. The numbers seen when placing the GPU in slot 4, are similar to numbers we have seen on another X79 motherboard, and therefore could be related to the chipset, and possibly also that the GPU is running in 8–lane mode, instead of 16–lane mode.

The full charts for each of the PCIe slots are included in section B.1 on page 77. Particularly interesting are figure B.2 and B.4. In these charts, we can see that the write bandwidth stops growing abruptly. In all the other charts, the bandwidth grows steadily and slowly flats out. Why this is happening is a little unclear, but it is likely that there is a bug or bottleneck in the chipset or CPU.

Current vs. previous generation chipset

We have also run tests to compare the current generation X79 chipset versus the previous generation chipset X58 chipset. This is an interesting benchmark, as there possibly are bugs present in the chipsets affecting peer–to–peer communication. Figure 4.3 shows DMA push bandwidth and figure 4.4 shows DMA pull bandwidth, both charts are on the next page.

Looking at the first figure (4.3), we can see that the write performance on X79 grows comparable to the X58 chipset up to transfer sizes between 8 and 16 kb. With transfer sizes above that, the bandwidth seems to top out around 2650 MB/s. This does not happen on X58, where the bandwidth continues to grow with larger transfer sizes.

Read performance is also affected, as seen in the second figure (4.4), and both for X58 and X79 the read bandwidth is significantly lower than what can be achieved through PCIe switches. The read performance on X79 is also lower than on X58, but it does not appear to top out as abruptly as it does with write performance. It just grows slower and gradually tops out at just above half the bandwidth achieved on X58.

Write speed on X58 is comparable to using a PCIe switch, but read performance is significantly lower. On X79, the difference is even larger, with both write and read performance being significantly lower.

With the X99 chipset from Intel coming soon, it will be interesting to see how that compares, as there is evidence that the performance is greatly affected by the chipsets.

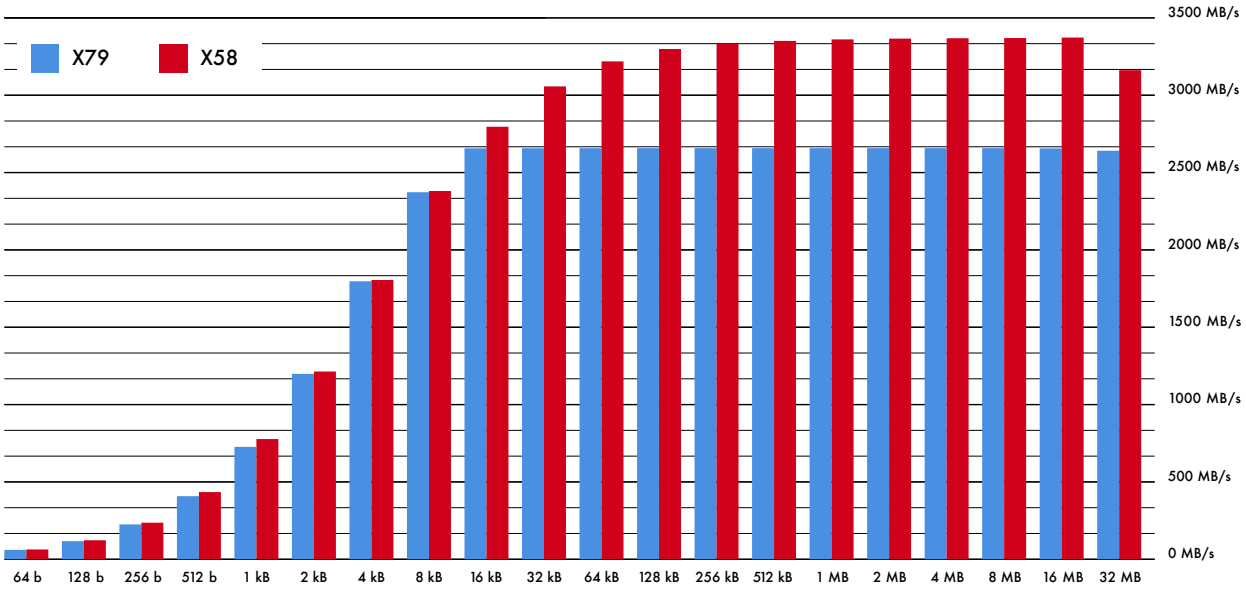


Figure 4.3: This figure shows bandwidth when performing a DMA write to a GPU via the CPU chipset. The blue bars are with the current generation X79 chipset, the red bars are with the previous generation X58 chipset.

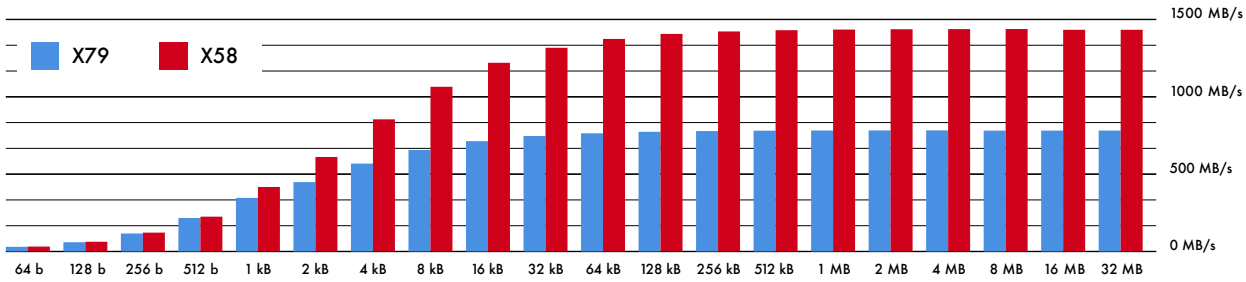


Figure 4.4: This figure shows bandwidth when performing a DMA read from a GPU via the CPU chipset. The blue bars are with the current generation X79 chipset, the red bars are with the previous generation X58 chipset.

4.2.2 Pipeline benchmarks

While the synthetic benchmarks are useful for detecting potential bottlenecks, they are not representative for how actual applications behave. In our pipeline, transfers usually occur in bursts, up to 50 times per seconds, depending on the frame rate, and multiple simultaneous transfers are also common. This is far from the constant, but never concurrent transfers that are benchmarked with the synthetic tests. We have therefore also benchmarked the individual parts of our pipeline, and we focus on the data transfers and the frame synchronizer as that are the parts we have implemented for this thesis.

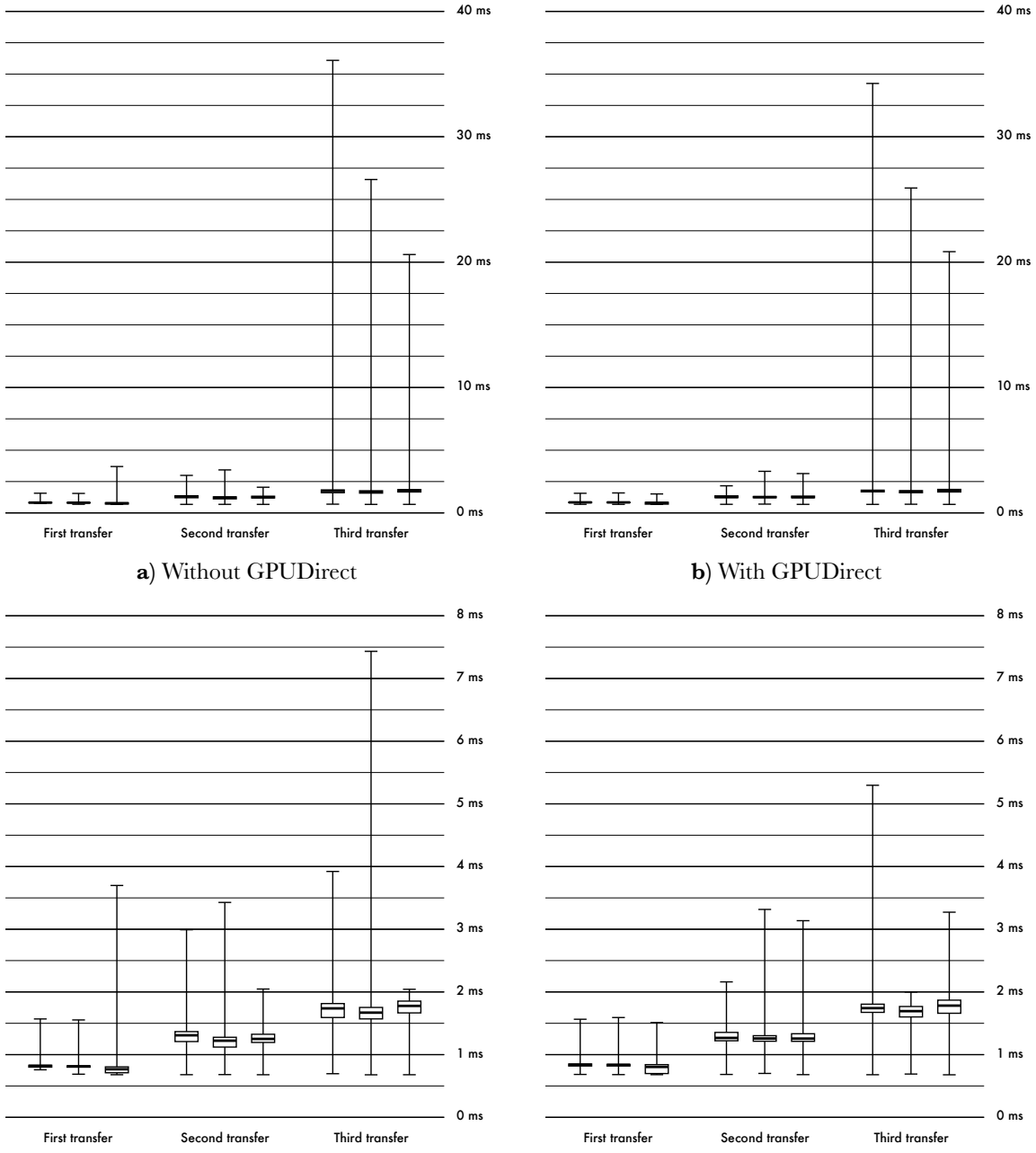
Pipeline DMA latency

The first thing we have benchmarked in the pipeline, is DMA latency. This is done by saving a timestamp right before and right after we start each DMA transfer. By subtracting the first timestamp from the second one, we get the latency of the transfer, including overhead. Figure 4.5c on the next page shows the latency at different frame rates. From this figure we can see that the first of the started DMA transfers have the lowest latency. This is expected, as the first transfer will have the entire link to itself for at least some of the total transfer time. For the second transfer, we see that the latency has increased. This is also expected, as this transfer often starts in parallel to the first transfer and finished in parallel to the third transfer. The third transfer has the highest latency of all transfers. There are two reasons for this. First, the Dolphin IX cards have two DMA engines. This means that the third transfer will have to wait for one of the two other transfers to complete, before it is started. Second, the third transfer will often run in parallel to the second transfer.

In addition to the numbers in figure 4.5c, we have looked at latency numbers when using GPUDirect. These can be seen in figure 4.5d. It shows that the latency, when using GPUDirect, is almost identical to the latency without GPUDirect. Compared to the results we have seen from `dma_bench` tests, this is very similar. We have seen that under optimal conditions, the transfer latency of GPUDirect transfers are comparable to the latency when transferring between CPU memory segments.

Under these conditions, where transfers with and without GPUDirect have the same latency, using GPUDirect will always be preferable. The extra time spent copying data from CPU memory to GPU memory, required without GPUDirect, will always add latency. While this is true for write operations, it is not guaranteed to be correct for read operations, as we have seen significantly higher latency for read operations.

Depending on how we have configure the pipeline, it may also have a second transfer. For testing this, we have configured the pipeline as described in section 3.7.1 on page 48. In this setup, the stitched image is sent to a separate machine for encoding. The resolution of the image where this transfer occurs in the pipeline, is 2872×1864 pixels when running with three cameras. Between these modules YUV422 is used. In total, the size of each frame is 10.71 MB. While this is larger than each of the images in the first series of transfers, there is only one transfer. Under normal conditions, the pipeline is fast enough that this transfer is likely to occur independently of the transfers from recording machine to processing machine.



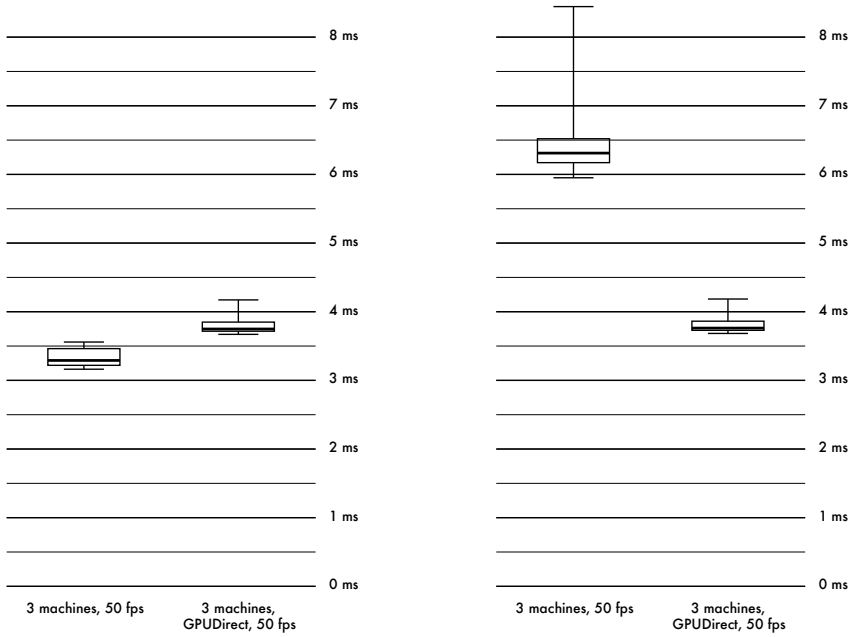
a) Without GPUDirect

b) With GPUDirect

c) Without GPUDirect (spikes removed)

d) With GPUDirect (spikes removed)

Figure 4.5: DMA latency of the individual frame transfers between the recording machine and the processing machine. Values are grouped by the order of the transfers. The boxes in each group are, from left to right: 30 fps, 50 fps and 50 fps. **a**) and **b**) show latency without and with GPUDirect. We have confirmed with Dolphin that there is a bug affecting more than two parallel DMA transfers. Because of this, some of the transfers in the third group take much longer than expected to complete. To better see the variation in results not affected by this bug, we have included two additional charts, **c** and **d**. These charts show the same results, with spikes above 15 ms removed. The whiskers show smallest and largest value recorded.



a) Isolated DMA latency.

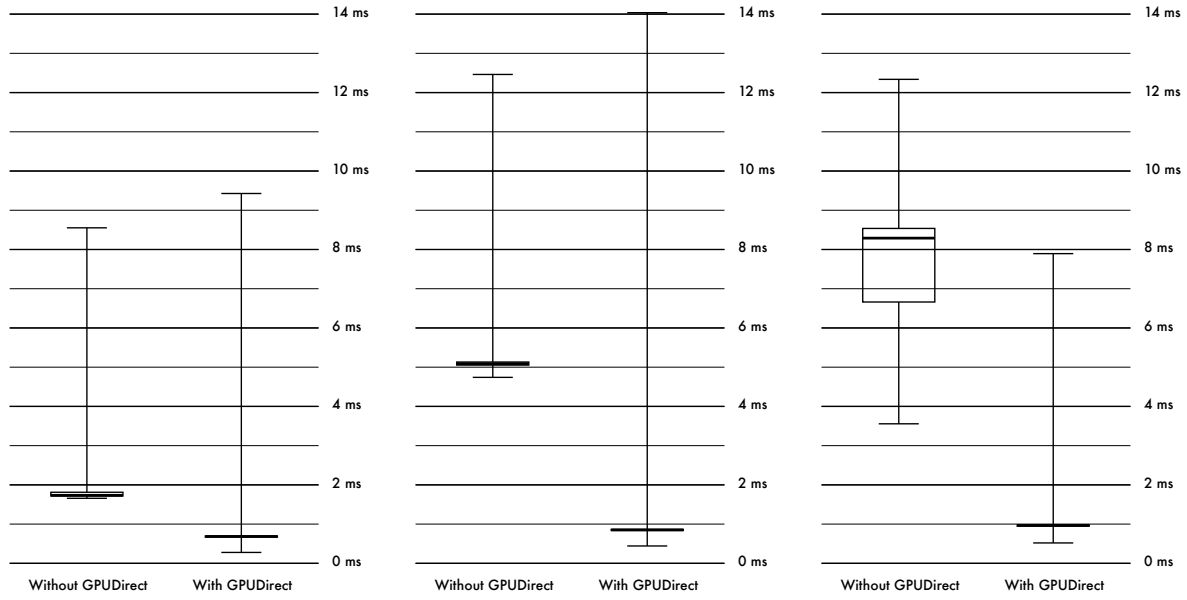
b) Total transfer latency

Figure 4.6: Latency introduced by running panorama processing and encoding on separate machines. This corresponds to figure 3.16 on page 48. **a**) shows the isolated latency of the DMA request. **b**) shows the total latency from processing has completed, until the frame is ready for encoding. Without GPUDirect this includes a memory copy from GPU to CPU. The whiskers show smallest and largest value recorded.

The latency of this transfer is shown in figure 4.6a above. When running with GPUDirect, this is a direct DMA transfer from GPU memory on the processing machine to CPU memory on the encoding machine. Without GPUDirect, it is a normal DMA transfer from CPU memory in the processing machine to CPU memory in the encoding machine. We can observe that when using GPUDirect, this transfer takes slightly longer than without. Based on the numbers we have seen in figure 4.2, on page 58, that these results are as expected.

The real gain from using GPUDirect is not lower latency transfers, it is the ability to remove the initial copy from GPU memory to CPU memory on the processing machine. Figure 4.6b compares total transfer time with and without GPUDirect. When not using GPUDirect, this includes the copy from GPU memory to CPU memory before copying from machine to machine. This chart clearly shows that the slightly higher latency of the DMA request is insignificant compared to the extra latency added by the additional copy from GPU memory to CPU memory without GPUDirect.

By dividing the number of bytes transferred by the total time the transfer took, including the extra step via CPU memory when not using GPUDirect, we get a median bandwidth of 2846 MB/s with GPUDirect and 1696 MB/s without. This is an interesting result. At 1696 MB/s, going via the CPU is still faster than using GPUDirect under sub–optimal conditions. As figure 4.2 (p. 58) shows, we can not expect bandwidths above 800 MB/s when the communication



a) Three cameras **b**) Seven cameras **c**) HDR mode

Figure 4.7: Total latency added by the frame synchronizer. **a**) shows latency with three cameras. **b**) shows latency with seven cameras. **c**) shows latency in HDR mode with three cameras. All measurements are at 50 fps. The whiskers show smallest and largest value recorded.

between the Dolphin IX card and the GPU goes via the CPU on X79 chipsets. On X58, as seen in figure 4.4 (p. 60), using GPUDirect will likely be slightly slower.

Frame synchronizer latency

After transferring the frames from the recording machine to the processing machine, the next step in the pipeline is frame synchronization. This is a step that largely consists of memory copies. Frames from the individual cameras are grouped based on their timestamps and copied into common buffers for the sets. After frames have been grouped into sets, each set is copied into the buffer provided by the next module in the pipeline. When running with GPUDirect, these copies are performed in–device on the GPU. Without GPUDirect, all happens on the CPU.

Depending on the number of cameras, the memory copies performed in the synchronizer can potentially be a source of delays. Each frame is 2.10 MB. With five cameras, 21 MB are copied for each frame set. At 50 fps, this is marginally more than 1 GB/s.

Figure 4.7 above shows the latency added by the frame synchronizer under different conditions. On the left, figure 4.7a, the latency with three cameras are shown. In the center, figure 4.7b shows the latency with seven cameras. On the right, latency when running in HDR mode is shown in figure 4.7c. All numbers shown here were recorded with the cameras running at 50 fps. Large figures with additional data are included in section B.2 on page 81.

Cameras	Without GPUDirect		With GPUDirect	
	Median	Maximum	Median	Maximum
3	7.62	7.98	19.31	47.63
7	6.06	6.51	36.18	69.74

Table 4.5: Approximate data throughput of the frame synchronizer, calculated by dividing the time spent by the number of bytes copied. The numbers without GPUDirect are from an i7-4820K and the numbers with GPUDirect are from a Tesla K20c. All numbers are in GB/s.

All figures show that the frame synchronizer is substantially faster when using GPUDirect. We also see that as more data is being copied, the advantage of using GPUDirect is growing. This is clearly visible in figure 4.7b where the cost of adding four more cameras compared to 4.7a is insignificant when using GPUDirect. Without GPUDirect, the time spent in the synchronizer is more than doubled.

Figure B.8 on page 81 clearly shows how the synchronizer is affected by the size of transfers. At normal frame rates, where each frame is slightly above 2 MB, the difference with and without GPUDirect is around 1 ms. When using HDR, each frame is just over 4 MB, and we see a larger difference between using GPUDirect and not using GPUDirect.

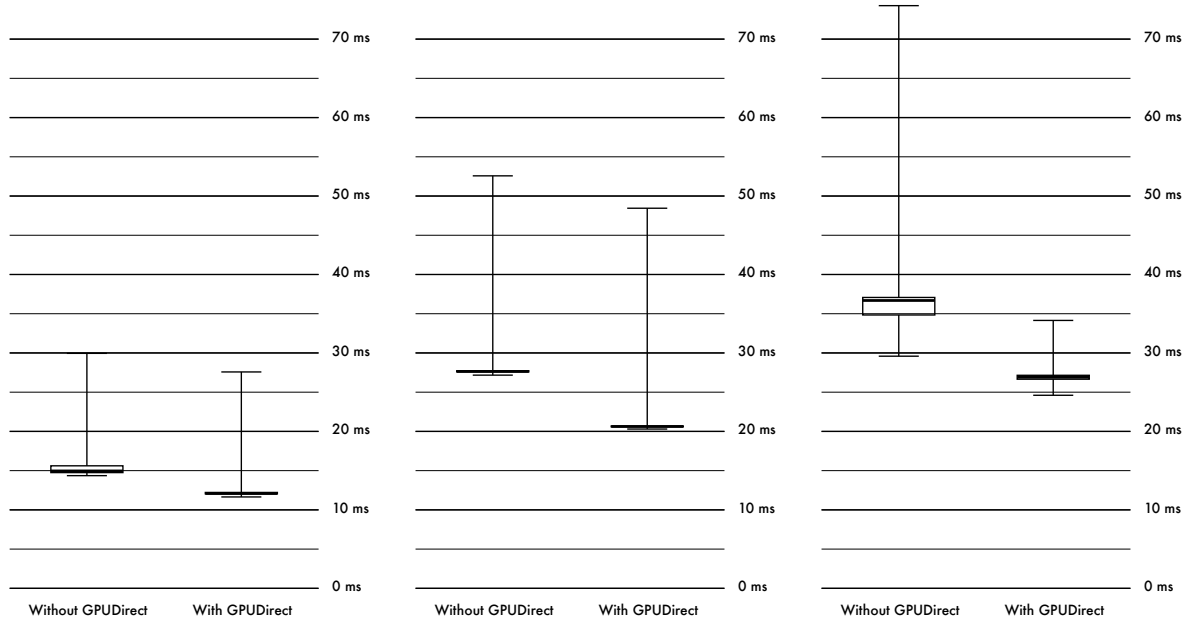
The reason the frame synchronizer is quicker when using GPUDirect is most likely because memory bandwidth on GPUs is significantly higher than memory bandwidth on CPUs. The only significant difference between running the frame synchronizer with and without GPUDirect is the location of the frame data.

The i7-4820K has a maximum memory bandwidth of 59.7 GB/s. The two GPUs we have used for testing, the Quadro K2000 and Tesla K20c, have 64 GB/s and 208 GB/s of bandwidth respectively. From actual measurements, we have seen that the Quadro has 25 GB/s of bandwidth, and the Tesla has 83 GB/s. The i7-4960X [66] has a measured bandwidth of around 40 GB/s [67]. According the the specifications from Intel, the i7-4820K and i7-4960X has the same maximum bandwidth. All the tests shown here have been run with the Tesla K20c card.

It is also possible to perform copies in GPU memory without using GPUDirect, by letting the frame synchronizer perform the copy from CPU memory to GPU memory before synchronizing the frames. This removes the need for a separate module to handle the copy from CPU memory to GPU memory. It also reduces latency by utilizing the increased memory bandwidth on the GPU.

Table 4.5 above shows the data throughput of the frame synchronizer. The throughput on the CPU is significantly lower than on the GPU. As the frame synchronizers operate on a single thread, all memory transfers are performed sequentially. Particularly on CPU, memory bandwidth appears to be significantly reduced because of this. We also observe that larger transfers result in lower throughput on the CPU and higher throughput on the GPU. This can possibly be because the CPU operations can be interrupted by context switches.

In addition to the higher latency, the measurements with GPUDirect are also more stable. This is clearly visible in the full charts in section B.2 (p. 81), particularly in figure B.8. This figure also shows that the latency varies greatly between frame rates. It is unclear why this is happening,



a) Three cameras **b**) Seven cameras **c**) HDR mode

Figure 4.8: Total latency of the processing pipeline when running on two machines. **a**) shows latency with three cameras. **b**) shows latency with seven cameras. **c**) shows latency in HDR mode with three cameras. All measurements are at 50 fps. The whiskers show smallest and largest value recorded.

as there is nothing in the frame synchronizer that changes based on the frame rate, except for dropped frame deadlines. This is supported by the fact that the minimum latency of each test is the same, independently of frame rate.

Another interesting find in figure 4.7c (p. 64) is that the frame synchronizer has higher latency in HDR mode than with 7 cameras. This is unexpected, as more date is copied with seven cameras.

Processing pipeline latency

N.B.: *The latency numbers presented in this section are not representative of the total latency of the pipeline. When not otherwise stated, the latency is measured from the last of the individual frames are received on the recording machine until the stitched image is ready for encoding. The actual latency from the image is captured until it is encoded is larger. We have chosen to measure latency this way, because it is possible to compare across configurations and is independent of camera trigger synchronization, which can be problematic (see figure B.12 on page 85).*

There are many factors that can affect the total processing pipeline latency. As we have seen in the previous sections, the frame synchronizer and data transfer modules are affected by the amount of data. The data size can vary because of either different formats or the format used. Other parts of the pipeline always read frames in the same format, and are only affected by the number of cameras.

In figure 4.8 on the previous page we compare the latency of the pipeline with three different camera setups. Figure 4.8a shows a setup with three cameras, figure 4.8b with seven cameras, and figure 4.8c with three cameras in HDR mode. Latency for the rest of the modules in the pipeline with three and seven cameras can be seen in figure B.11 on page 84. All numbers are from a two machine setup, with the cameras operating at 50 fps.

Comparing the latency of these setups gives a good overview of how the pipeline scales. In table 4.4 on page 57 we listed the input and output size for each of these setups. From this we can see that the input size with seven cameras are 2.3 times larger than the input size with three cameras. The output size is 1.9 times larger. In HDR mode, the input is twice the size of three cameras, and the output is the same size.

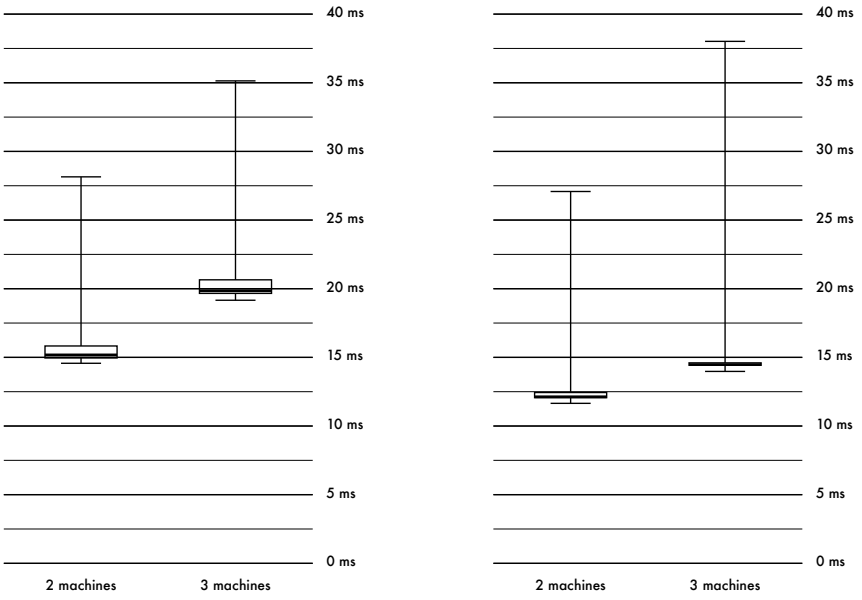
For each of the charts in figure 4.8, we can see that using GPUDirect yields lower latency. It is also apparent that, as the total latency increases because of addition data, the advantage of using GPUDirect is growing. With three cameras, using GPUDirect reduces the overall latency with 2–3 ms. Looking at the big picture, including the latency of encoding the video, the latency of transfers, and image capture, this is insignificant. When increasing the number of cameras to seven or running in HDR mode, using GPUDirect reduces the latency with 7–8 ms. This is more significant, but the difference is still relatively small.

In the tests with a two machine setup, a large portion of the gains likely comes from the frame synchronizer performing memory copies on the GPU instead of the CPU. In figure 4.9, we compare the latency when distributing the pipeline across two and three machines. This is a situation where the latency of the pipeline is more reduced from using the GPUDirect technology itself, than from faster memory copies. With the pipeline setup used here, GPUDirect removes two copies between CPU memory and GPU memory. From figure B.11 we can see that these two copies take 4–5 ms with three cameras. The copy to GPU memory takes slightly more than 1 ms, and the copy back to CPU memory takes a little more than 3 ms. This corresponds to the difference with and without GPUDirect, as can be seen in figure 4.9a and figure 4.9b, being around 5 ms when running across three machines.

We also observe that the three machine setup with GPUDirect has slightly lower latency than the two machine setup. Even with a frame synchronizer that performs memory copies on the GPU, the latency of a three machine setup with GPUDirect will likely have less than 1 ms more latency than a two machine setup without GPUDirect. This is because we, by using GPUDirect, can copy data between a GPU in one machine and memory in another machine almost as fast as we can copy the data from the GPU to CPU memory in the same machine.

In addition to the results we have presented in this section, we have tested multiple other pipeline configurations. Results from these tests are shown in section B.3 on page 83. Figure B.13 on page 86 shows the processing latency of all three camera setups we have benchmarked. From this it is obvious that our GPUDirect setup yields lower latency than going via the CPU when transferring data between machines. With both two and the tree machine setups, GPUDirect is faster. The two machine setup is 2–3 ms faster, and the three machine setup is 5–10 ms faster.

Another interesting find is that while the delay measurements have large variance between the different frame rates of each setup, the minimum measured delay between the different frame rates is very similar. It also does not appear that the bandwidth affects the pipeline latency, as higher frame rates appear to have less variance than lower frame rates in some of the tests. This



a) Without GPUDirect **b**) With GPUDirect

Figure 4.9: The panorama processing pipeline latency when distributing the pipeline across two and three machines. The whiskers show smallest and largest value recorded.

could potentially suggest that there is some power saving feature or similar at work, affecting the performance as the workload decreases. It would be worth turning off power saving features, and running these tests again and see if the variance is equally large.

4.3 Evaluation

The current prototype setup at Alfheim Stadion is a three machine setup, with two recording machines and a single processing machine. This setup was a result of the available machines and limitations imposed by these. No more than four cameras could be connected to each of the recording machines, and only the processing machine had enough PCIe lanes left for a powerful GPU. With new machines purchased specifically for the purpose of running the distributed pipeline, we have more flexibility.

The new recording machines have more PCIe slots. By installing additional ethernet adapters in these machines, we can run more cameras per machine. In our tests, we have run seven cameras on a single recording machine without problems. With the K20c GPU from Nvidia, we can also run the processing pipeline on a GPUDirect capable device.

Distribution latency

The latency added by distributing our system is the result of up to three separate steps, depending on how the pipeline is configured. If we split it between two GPU modules, three steps are added, between a CPU and a GPU module, two steps are added, and between two CPU modules a single

	CPU to GPU	GPU to CPU	Dolphin IX	Dolphin IX GPUDirect
Theoretical	8 GB/s	8 GB/s	4 GB/s	4 GB/s
Actual	5–6 GB/s	3 GB/s	3 GB/s	2–3 GB/s

Table 4.6: Theoretical and approximate median bandwidth of the steps added by distributing the pipeline. The actual numbers are calculated from latency measurements in our pipeline.

Data flow	CPU to GPU	GPU to CPU	Dolphin IX	Dolphin IX GPUDirect
CPU to CPU			×	
CPU to GPU	×		×	
GPU to CPU		×	×	
GPU to GPU	×	×	×	
GPUDirect				×

Table 4.7: Steps required to distribute the panorama based on the source and destination of the data.

step is added. With GPUDirect, all of these distribution configurations can be performed in a single step. The bandwidth for each of these steps are shown in table 4.6 and table 4.7 above shows which steps are required by the different configurations.

In table 4.8 below, we have calculated the approximate latency added by the different distribution configurations.

GPUDirect in the pipeline

Looking at table 4.8, we clearly see that the gains from using GPUDirect are highly dependent on the data source and destination. For example, going from CPU memory to GPU memory with GPUDirect only reduces the latency by 1.5 ms. On the other hand, going from GPU memory to GPU memory the, latency is less than half.

Data flow	Latency
CPU to CPU	3.5 ms
CPU to GPU	5.5 ms
GPU to CPU	7.0 ms
GPU to GPU	9.0 ms
GPUDirect	4.0 ms

Table 4.8: Latency added to the pipeline by distributing it between modules where the frames are 10.7 MB (3 camera panorama in YUV422).

With our pipeline running in real–time on a two machine setup, it is not necessary to add additional machines at the current scale. The machine–to–machine communication in this setup, is from CPU memory to GPU memory. This is a setup where the gains from using GPUDirect is minimal. Considering the high price of hardware that supports GPUDirect, using GPUDirect in this setup is not economically sensible. It is, however, useful to have the GPUDirect support implemented for future extensions to the pipeline.

The numbers used here to calculate the latency of GPUDirect, are based on optimal conditions. As we have seen in this chapter, the performance of GPUDirect is highly dependent on the PCIe layout. Under optimal conditions, the performance is on par with or slightly lower than CPU to CPU performance. With sub–optimal conditions, the performance can be so low that not using GPUDirect can be quicker. In our pipeline, this means that using GPUDirect to transfer data from a GPU in our current processing machines likely will be slower than not using GPUDirect.

4.4 Summary

In this chapter, we have shown that our distribution setup is working, and that the panorama processing pipeline is running in real–time. We have benchmarked individual parts of the pipeline, as well as the overall performance. This has helped us identify both opportunities for additional reductions in latency, and bottle necks. We even found and reported a bug in Dolphin's drivers.

In addition to being able to run in real–time, we have seen that the system is capable of running with more cameras than the current prototype setup. We have tested the setup with seven cameras, and this was no problem for our current setup. It will most likely be able to run with many more cameras. Unfortunately, we did not have more cameras available to test this.

With GPUDirect support, we can remove unnecessary steps. We have seen that this can reduce latency, and also that the scaleability of the system with GPUDirect is better than without.

Chapter 5
Conclusion

In this chapter, we summarize the work we have presented in this thesis. We also present our main contributions, and look at future possibilities of building on the work we have done.

5.1 Summary

In this thesis, we have reimplemented the Bagadus panorama processing pipeline. From running on a single machine, it is now distributed across multiple machines. To achieve this, we have gone through several steps.

We start out with an introduction of the Bagadus system in chapter 2. The system consists of multiple related components. The largest is currently the panorama processing pipeline, which is what we have worked on in this thesis. This pipeline consists of multiple steps, each with a different and limited role. Working together, these modules produce a panorama video from five cameras in real–time. We also discuss how the pipeline can operate autonomously, by automatically scheduling and start recordings. Other components of the system include a virtual camera viewer, player tracking, and user related features like a web interface for playback.

In chapter 3, we discuss how the panorama processing pipeline can be distributed across multiple machines. To make the pipeline more flexible, and to prepare it for distribution, we modularized it. The modules communicate with each other through a common interface we have designed. The data flow between the modules of the panorama is large, at some places larger than 1 GB/s. For sending these amounts of data between machines, we needed specialized equipment. We looked at several alternatives, and decided to use Dolphin IX equipment. The distribution of the pipeline was implemented with modules that integrate into the pipeline. This allows simple and flexible distribution setups. With a working distribution setup, we focused on minimizing the distribution latency and keeping resource usage at a minimum. We analyzed and optimized the data flow, removing unnecessary steps. This includes support for direct DMA transfers between GPUs in different machines, using Nvidia GPUDirect.

Finally, in chapter 4, we evaluated the performance of the distribution. We begin by looking at the performance of individual components. We did this to get an overall picture of what to expect from the pipeline. As the pipeline can be distributed in multiple different ways, we have evaluated two different pipeline setups which are representative for a range of possible configurations. With many possible configurations of the pipeline, some not possible or hard to test because of

hardware requirements, we used numbers from both component benchmarks and the pipeline measurements to get an indication of pipeline performance under different conditions and with different configurations. From the results found here, we identified possibilities for future improvements.

5.2 Main Contributions

As we discussed in section 1.2, we wanted to distribute the Bagadus panorama processing pipeline. We have implemented a prototype, showing that this is possible. By dividing the pipeline into separate modules, we made it easy to move steps of the pipeline around. All the modules in the pipeline communicate with the previous and next module through a common interface, which we have designed and optimized.

To distribute the processing across multiple machines, we designed and implemented modules that transfer data between machines. These modules use the same interface as the processing modules, allowing them to be inserted anywhere in the pipeline. The large data flow between machines require a high bandwidth link. We evaluated different interconnect solutions, and chose to use Dolphin Interconnect Solutions IX products. The modules for transferring data between modules are implemented with Dolphin's SISCI API.

To minimize the latency introduced by distributing the pipeline, we analyzed and optimized the distribution setup. This included implementing new modules for communicating between machines. The new modules improve upon the old modules, removing unnecessary steps and solving issues experienced with the initial modules. To further improve performance, we cooperated with Dolphin to implement support for GPUDirect in their drivers and APIs. With GPUDirect support in our pipeline, we can now send data directly from a GPU in one machine to GPU in another machine.

In addition to distributing the pipeline, we have also implemented a system to make the pipeline operate autonomously. We designed a system where schedule with recordings are stored in a database. To enter information into this database, we designed a simple web interface. This allows users to manually schedule recordings. We also created scripts to automatically fetch match schedules from external sources, and populate the recording schedule based on this. To operate autonomously, the pipeline runs all the times and automatically records, based on the schedule in the database.

We have also published two papers while working on this thesis:

Bagadus: An Integrated Real-Time System for Soccer Analytics

In this paper we introduce the new panorama image algorithm and the new distributed processing pipeline [9].

Interactive zoom and panning from live panoramic video

This paper describes the virtual camera viewer created to play back the panorama video [11].

5.3 Future work

At the end of chapter 3 we discussed further improvements that could be applied to the pipeline. There are still a couple of unnecessary memory copies related to the distribution. By distributing the synchronization step, and allowing the recording machines to perform the actual synchronization, we can possibly remove unnecessary transfers and memory copies.

The pipeline is currently limited to operating in serial. Some steps can be performed in parallel. For example, the encoding could be handled by two separate machines. As the video is stored in 3 segment files, each machine could encode three seconds. This would allow higher quality encoding, as it is no longer required that the encoder is performed in real–time.

In chapter 4 we identified areas where further optimizations could be applied. While the system is running in real–time without problems, there is still possible to reduce the processing latency.

The memory copies being performed in the frame synchronization is currently performed in CPU memory, even if processing is performed on GPUs. We identified that memory bandwidth on GPUs is much higher than on CPU. We could possibly speed up the frame synchronization step by copying the frames to GPU before performing any the actual synchronization.

We have also seen from our timing of the panorama pipeline, that the processing latency of various modules are not always stable. Since this is not a problem, because the steps rarely or never exceed their real–time deadline, it suggests that there is further room for improvements.

Appendix A

Accessing the source code

The source code of the current Bagadus pipeline is available at `https://bitbucket.org/mpg_code/bagadussii`. This repository includes the complete source of the project, including all modules discussed here. Access to this repository can be given upon request.

There are multiple branches. The branches which are related to the work presented in this thesis are `master` and `modular_pipeline`. The `master` branch is main branch. Most of the development is done there. The `modular_pipeline` branch is an experimental branch. This has modified build targets, to make it easier to compile different pipeline setups. Some of the modules that are no longer in use, like the first distribution modules, are stored in the `src/_backup/` directory.

Appendix B

Additional charts

B.1 DMA bandwidth of different PCIe layouts

In section 4.2.1 on page 57 we discussed DMA performance related to different PCIe layouts. The full charts from the `dma_bench` tests used to show difference there, are included here. The slot numbers in each figure description refer to the slot numbers in figure 4.1 on page 54.

N.B.: *The `dma_bench` tool used for benchmarking in this section runs multiple transfers per request. This results in reduced overhead and higher overall bandwidth. The different charts are comparable, but the numbers presented here are not comparable to other measurements.*

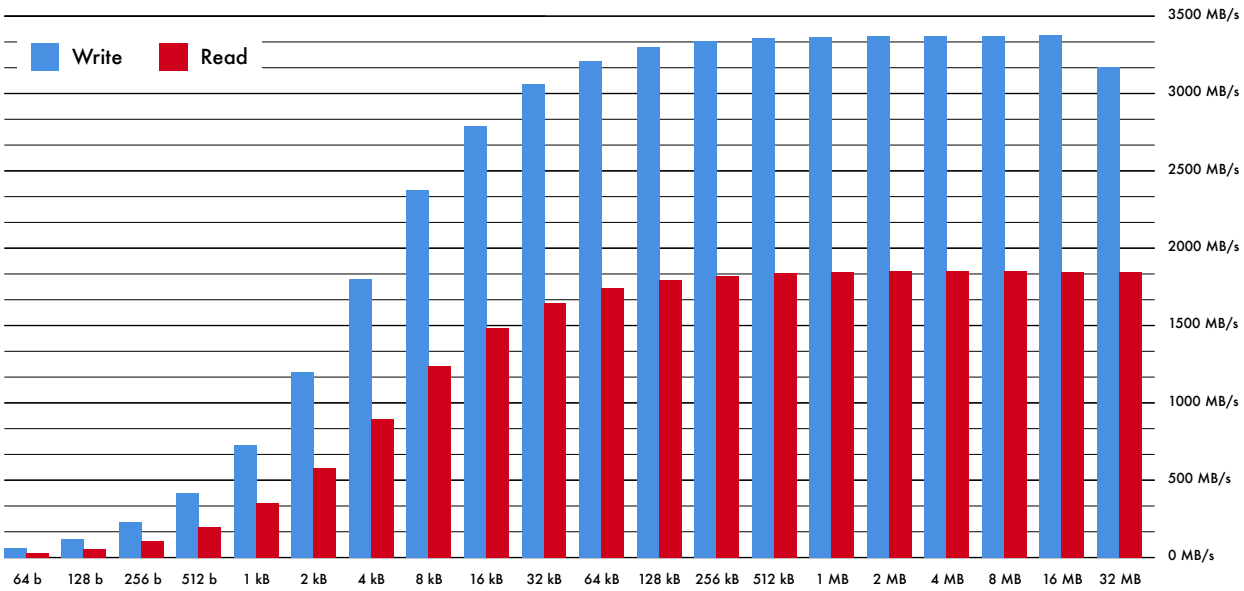


Figure B.1: DMA bandwidth with the GPU in PCIe slot 1 and the Dolphin IX card in PCIe slot 7

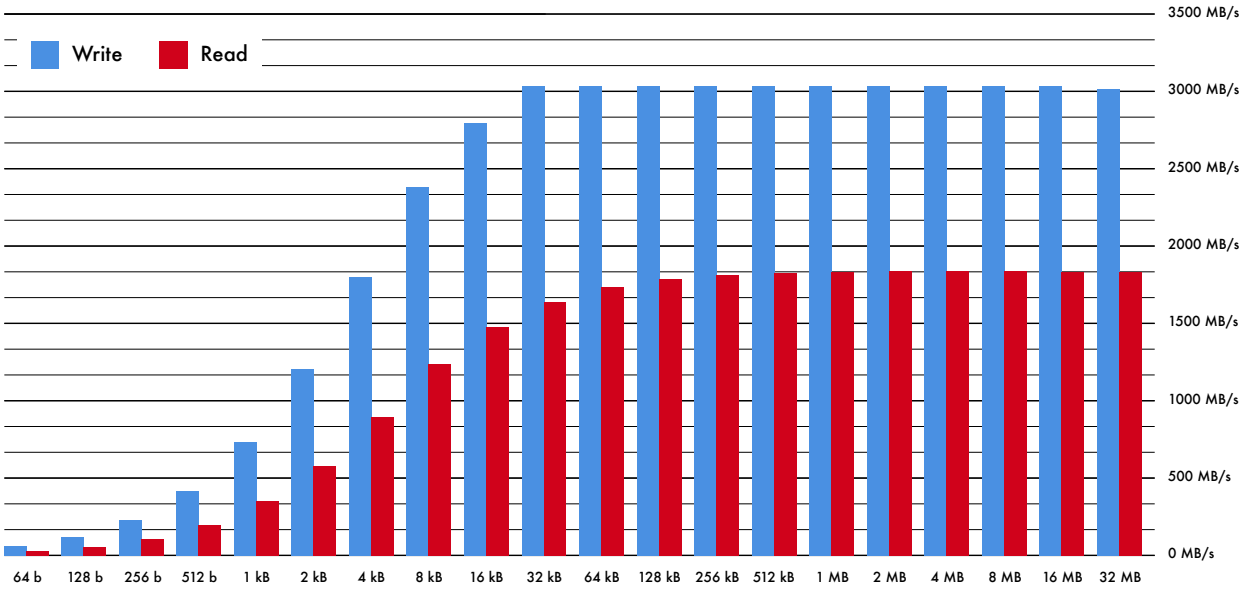


Figure B.2: DMA bandwidth with the GPU in PCIe slot 2 and the Dolphin IX card in PCIe slot 7

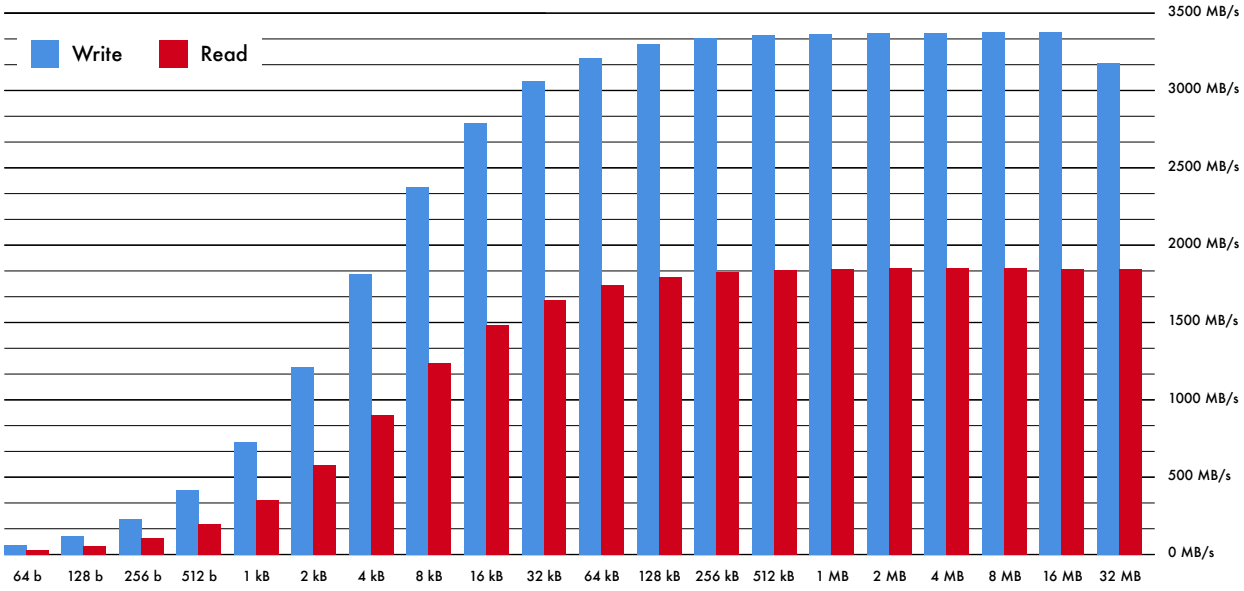


Figure B.3: DMA bandwidth with the GPU in PCIe slot 3 and the Dolphin IX card in PCIe slot 7

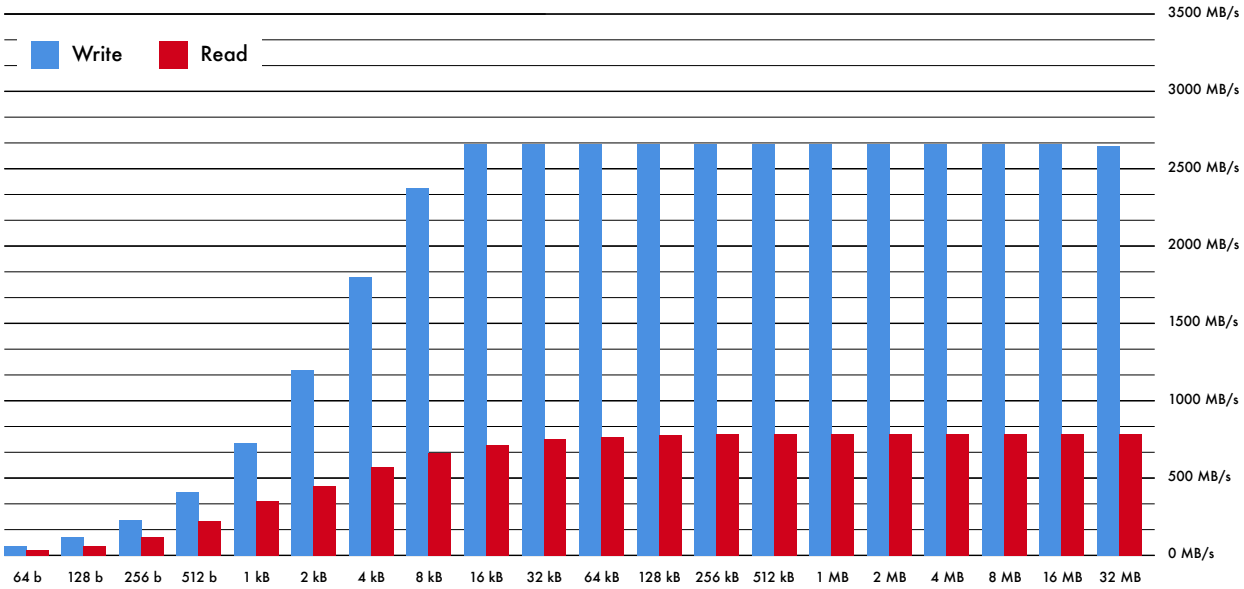


Figure B.4: DMA bandwidth with the GPU in PCIe slot 4 and the Dolphin IX card in PCIe slot 7

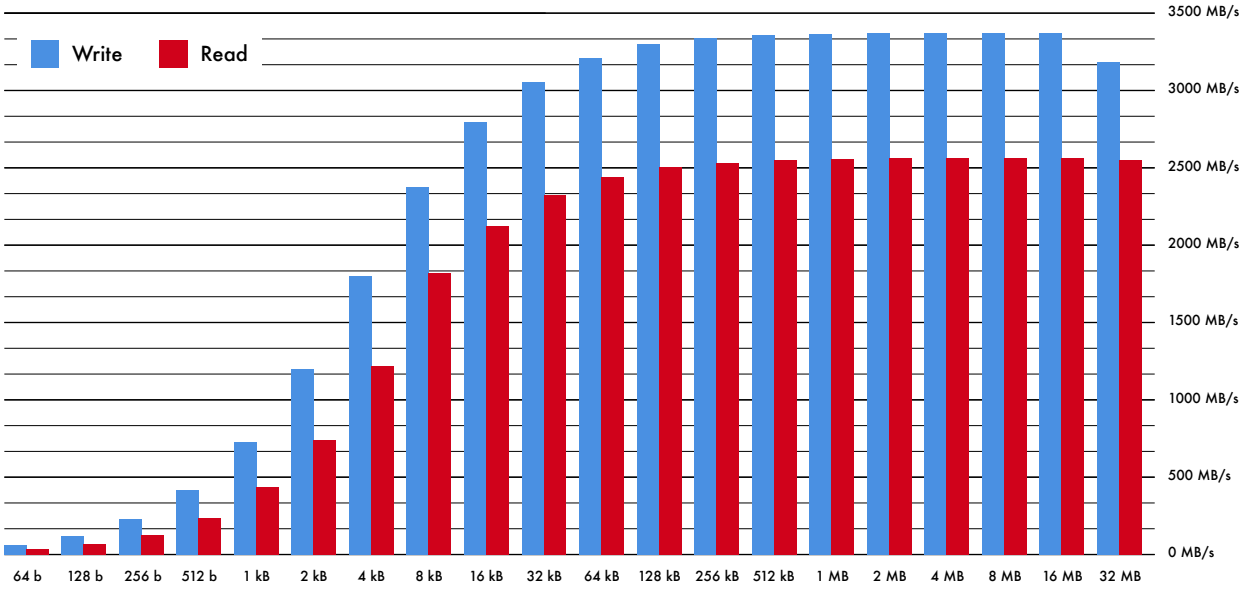


Figure B.5: DMA bandwidth with the GPU in PCIe slot 5 and the Dolphin IX card in PCIe slot 7

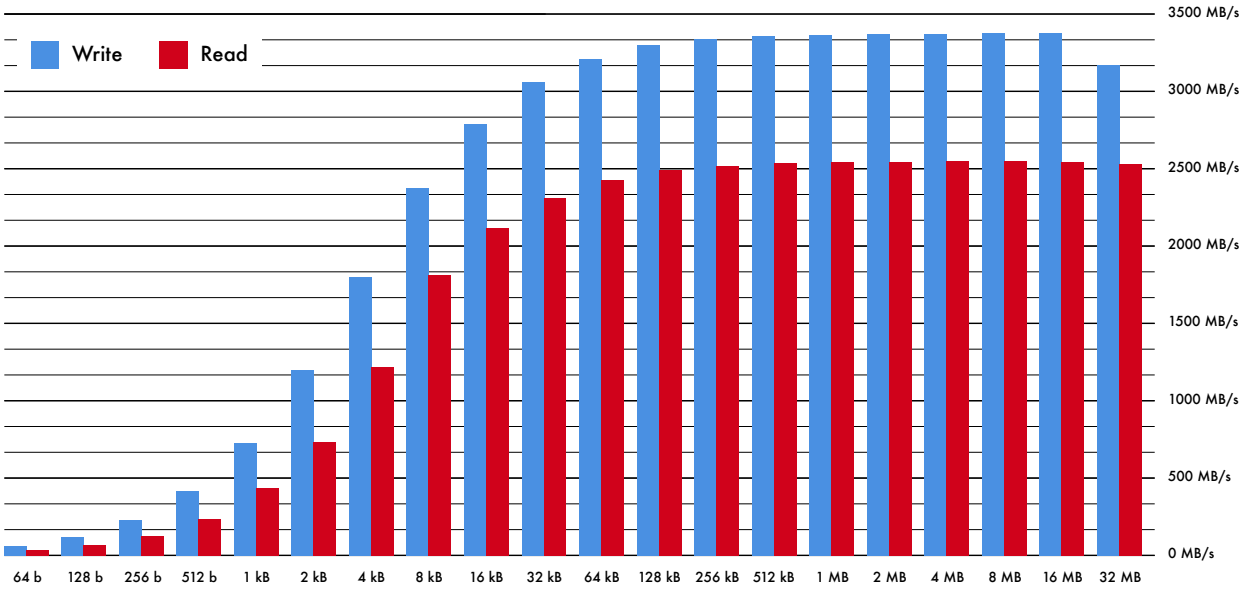


Figure B.6: DMA bandwidth with the GPU in PCIe slot 6 and the Dolphin IX card in PCIe slot 7

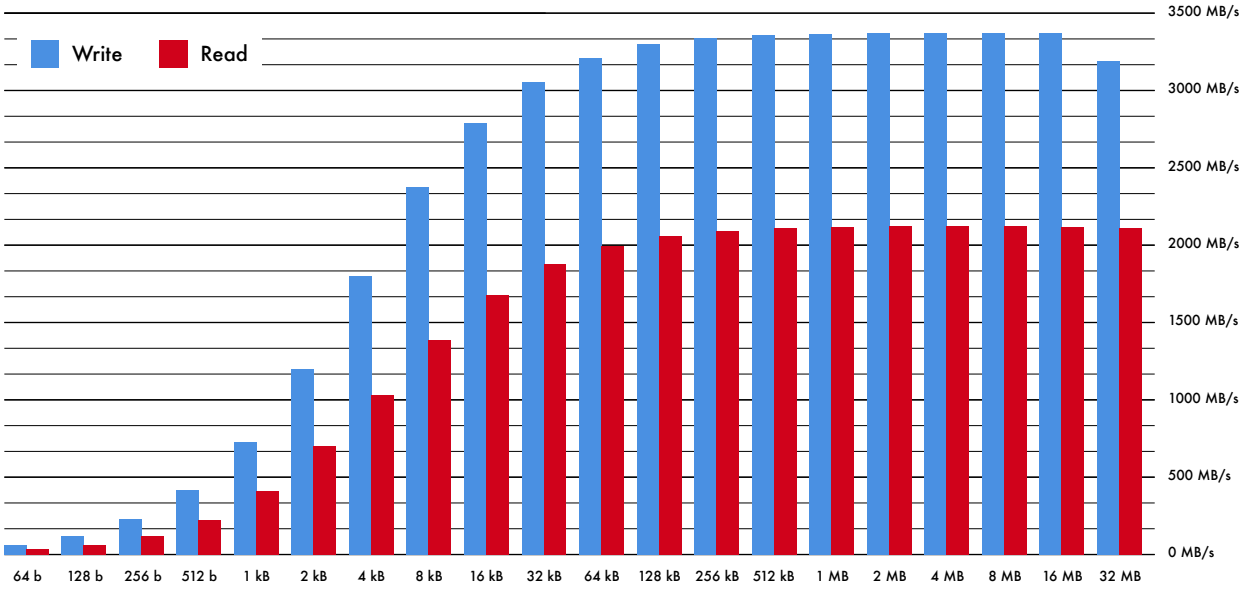


Figure B.7: DMA bandwidth with the GPU in PCIe slot 7 and the Dolphin IX card in PCIe slot 4

B.2 Frame synchronizer latency

We discussed the latency added by the frame synchronizer on page 64. Here we have included full charts with additional data points for different frame rates.

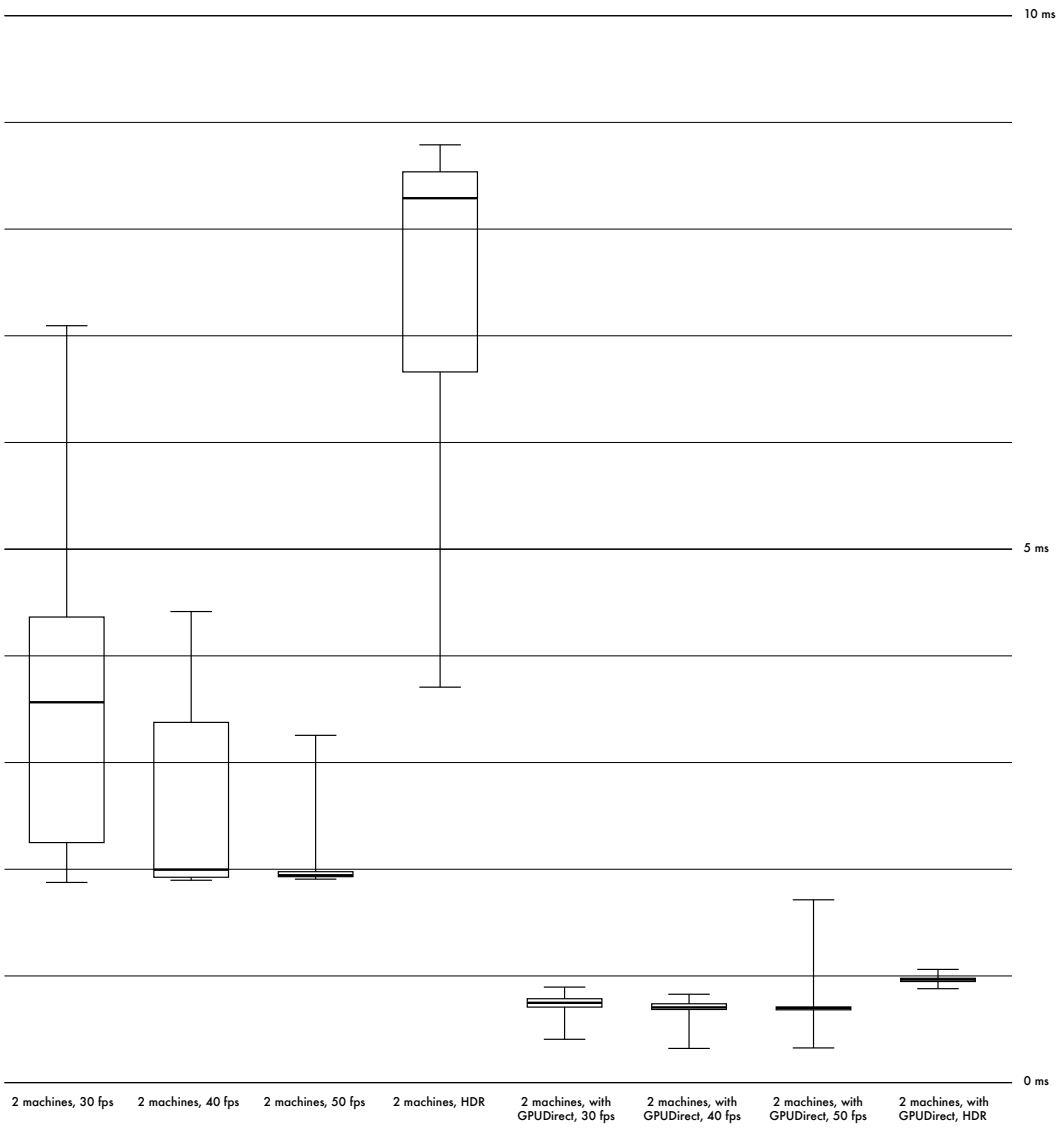


Figure B.8: The total latency added by the frame synchronizer at different frame rates. Latency is measured from the last frame transfer has completed, until the synchronized frame set is ready for processing. This does not include the memory copy from CPU to GPU when running without GPUDirect. The whiskers show smallest and largest value recorded.

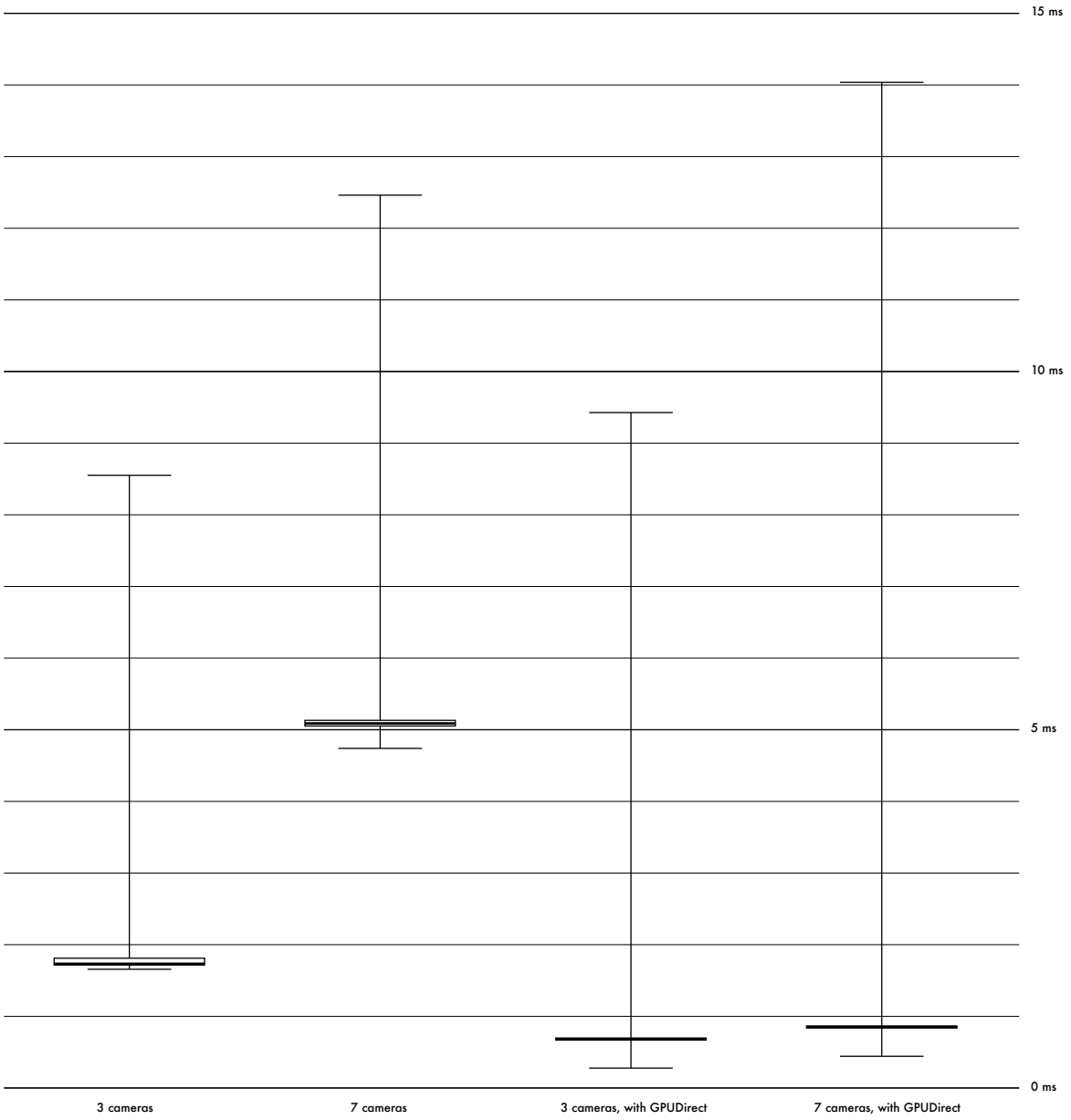


Figure B.9: The total latency added by the frame synchronizer with different number of cameras. Latency is measured from the last frame transfer has completed, until the synchronized frame set is ready for processing. This does not include the memory copy from CPU to GPU when running without GPUDirect. The whiskers show smallest and largest value recorded.

B.3 Pipeline latency

In addition to the charts included when we discussed the total pipeline latency in section 4.2.2, we have some additional and larger charts here.

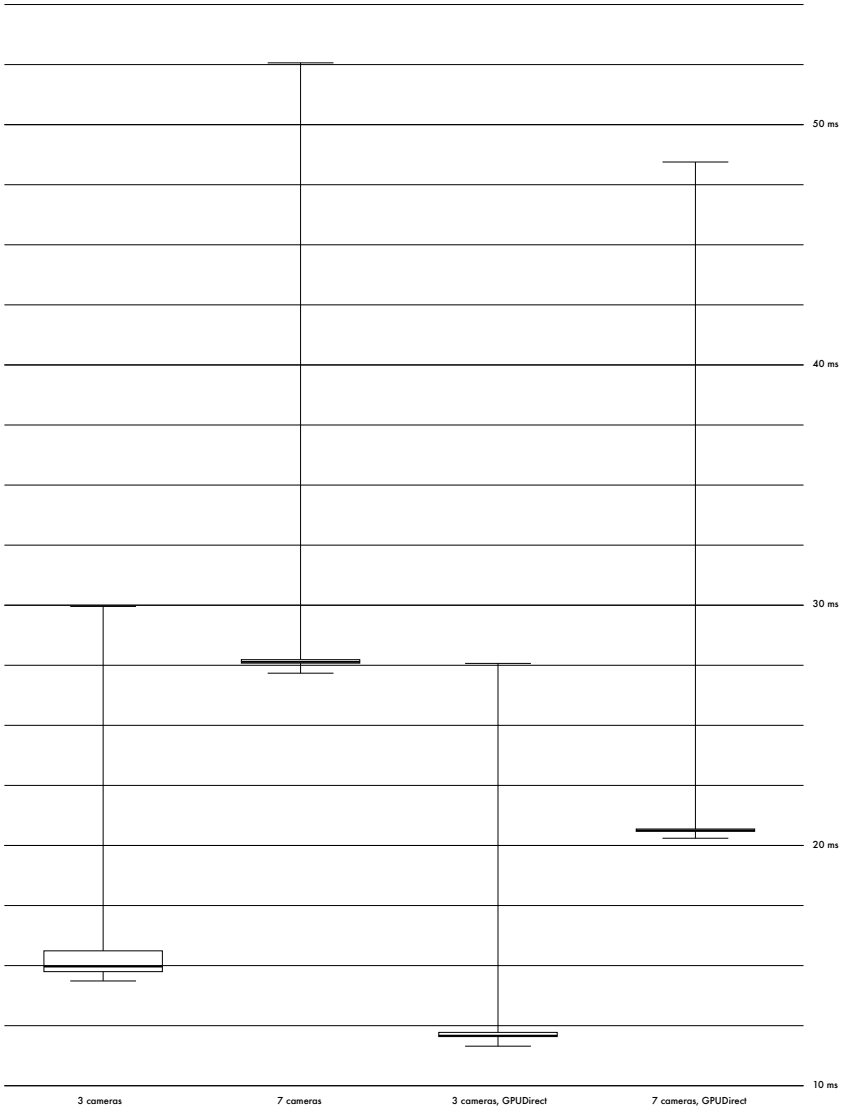
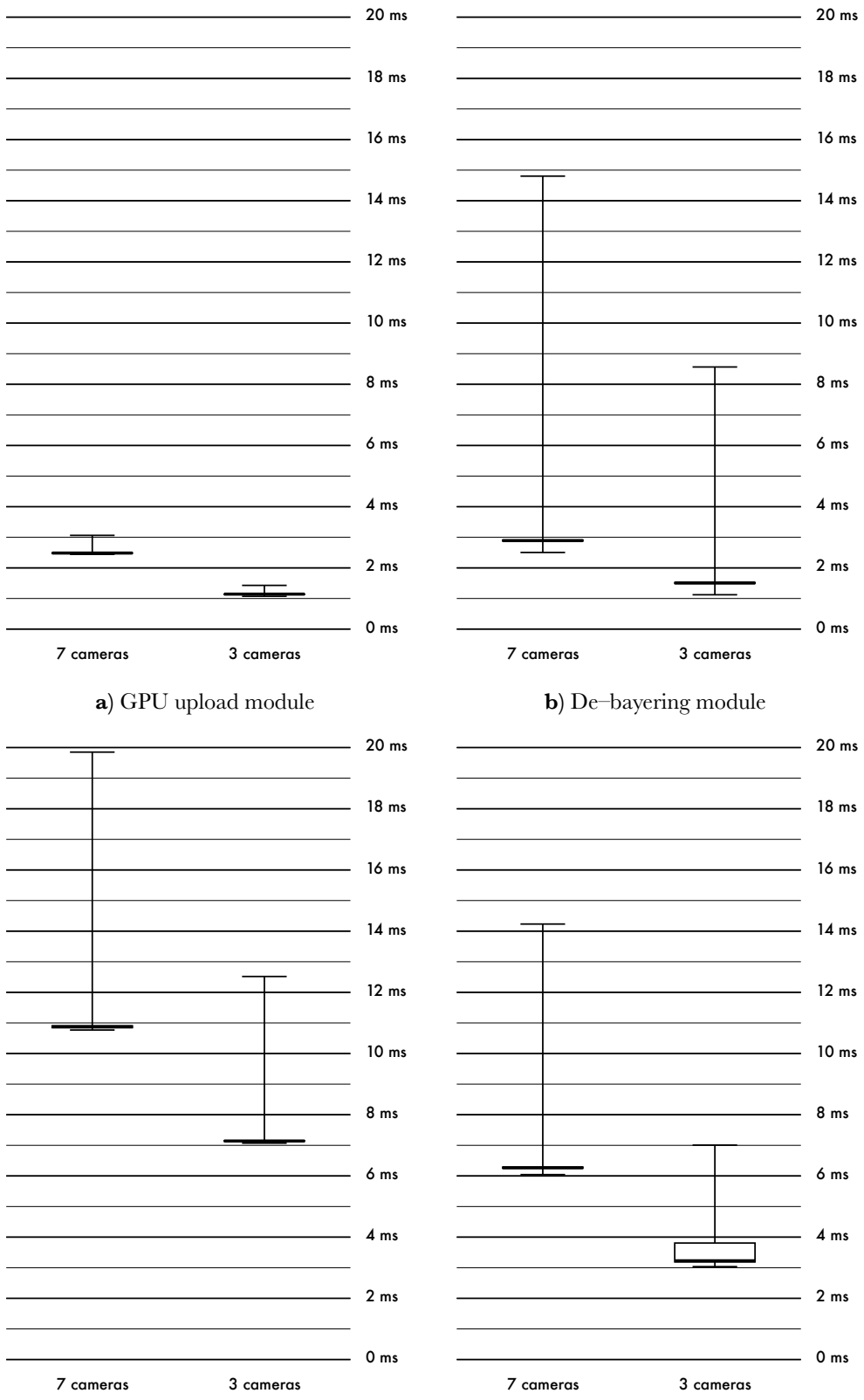


Figure B.10: The total latency added by the panorama processing with different number of frames. The latency is measured from the last frame transfer from the recording machine was completed, until the stitched image is ready for encoding. This does not include the latency added by the camera drivers or the H.264 encoding. The whiskers show smallest and largest value recorded.



a) GPU upload module

b) De–bayering module

c) Image stitcher module

d) GPU download module

Figure B.11: Individual module latency with three and seven cameras. The whiskers show smallest and largest value recorded.

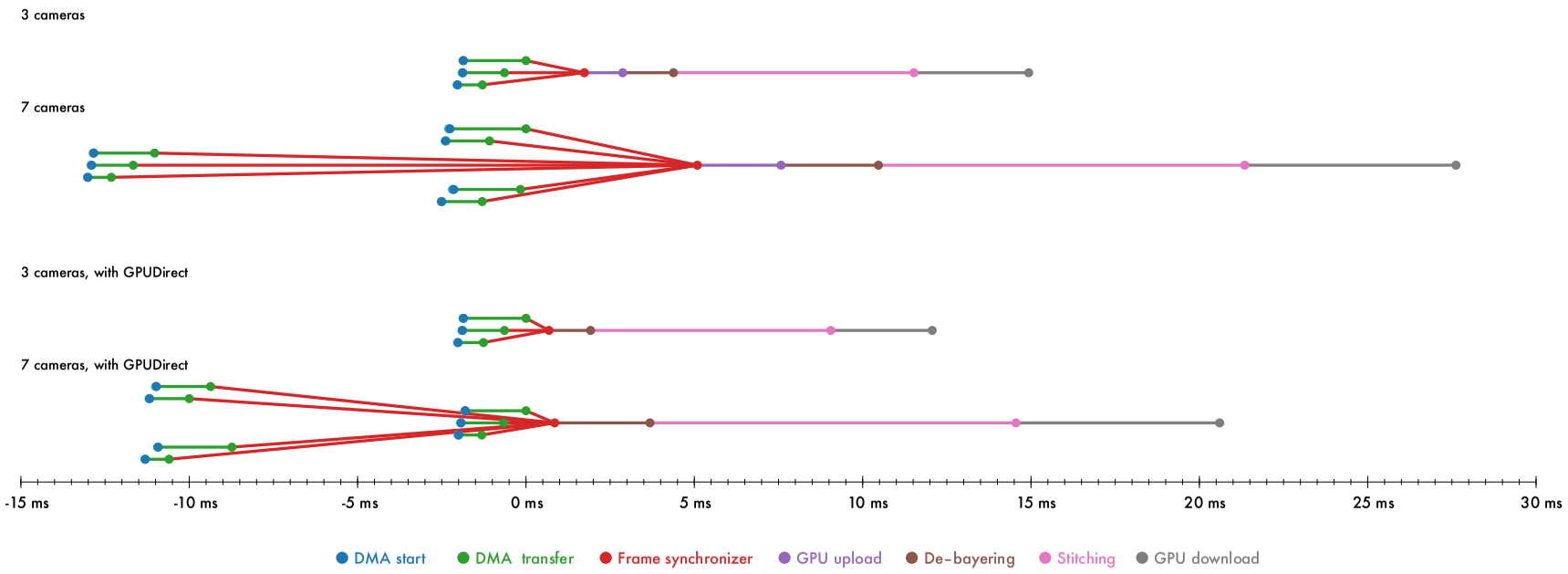


Figure B.12: Module timeline with three and seven camera setups. The timeline is centered around when the transfer of the last of the individual images was completed.

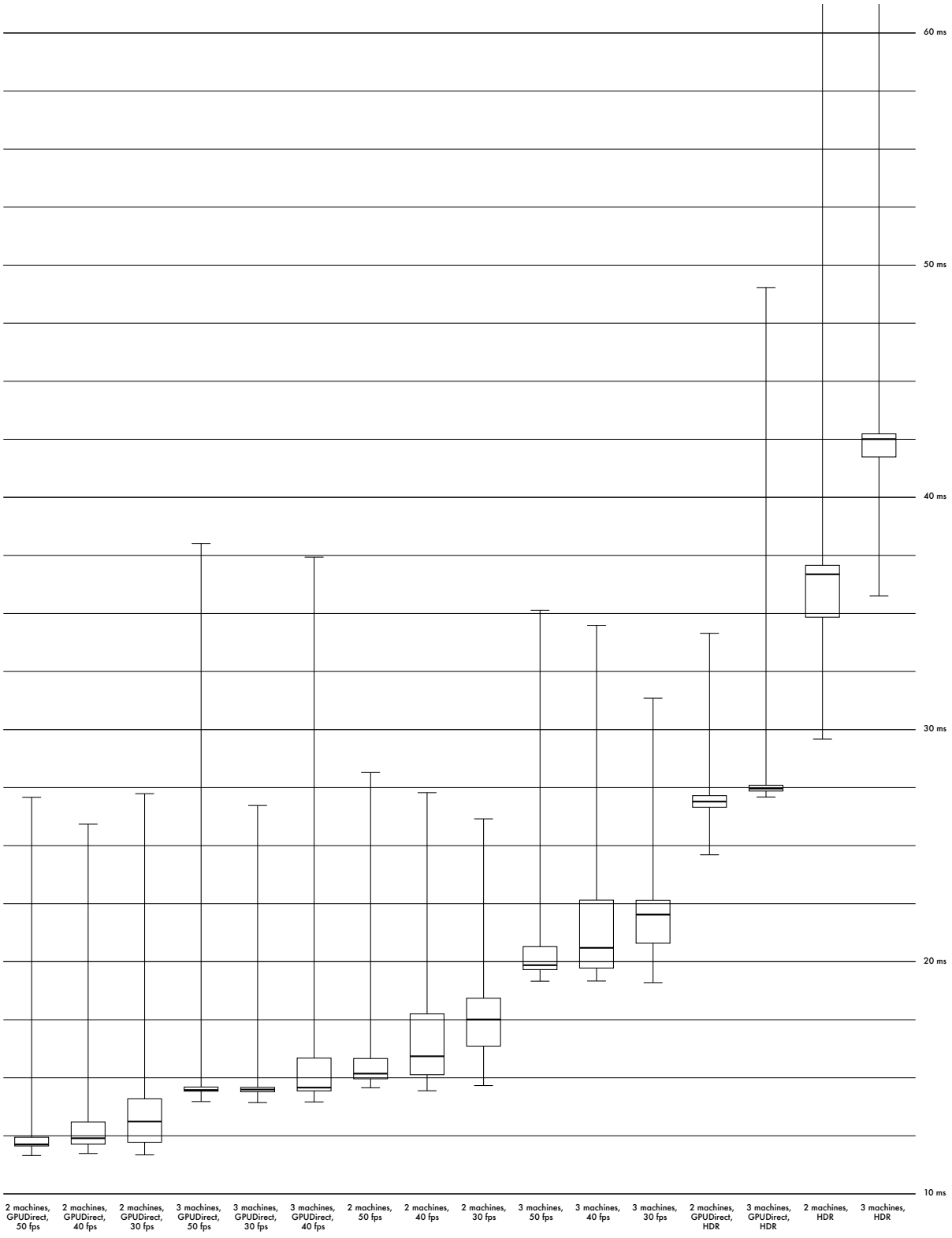


Figure B.13: The total latency added by the panorama processing at different frame rates and with different setups. The latency is measured from the last frame transfer from the recording machine was completed, until the stitched image is ready for encoding. This does not include the latency added by the camera drivers or the H.264 encoding. The whiskers show smallest and largest value recorded.

Appendix C

History of the Bagadus project

Version	Description	Contributors	References
0.1	Offline proof–of–concept	Simen Sægrov et al.	[13]
0.5	Offline viewer	Simen Sægrov et al.	[14]
1.0	Real–time single machine, enhanced panorama, dynamic stitching, color correction, background subtraction	Marius Tennøe, Mikkel Næss, Espen Oldeide Helgedagsrud, Henrik Kjus Alstad et al.	[9,15,68,69]
2.0	Modular design, distribution, automatic exposure, virtual viewer, HDR video, and position based event extraction	Asgeir Mortensen, Ragnar Langseth, Sigurd Ljødal et al.	[11, 70--73]
2.5	Improved distribution and upgraded camera modules.	Ragnar Langseth, Sigurd Ljødal, Asgeir Mortsensen et al.	[8]

Bibliography

[1] Militarizing Your Backyard with Python: Computer Vision and the Squirrel Hordes. https://www.youtube.com/watch?v=QPgqfnKG_T4. Accessed: 2014-07-04.

[2] GoalControl - Advanced Goal Line Technology. http://goalcontrol.de. Accessed: 2014-07-04.

[3] ZXY Sport Tracking AS. http://www.zxy.no. Accessed: 2014-06-11.

[4] NVIDIA. GPUDirect RDMA. http://docs.nvidia.com/cuda/gpudirect-rdma/index.html. Accessed: 2014-06-03.

[5] Espen Helgedalsrud. Efficient implementation and processing of a real-time panorama video pipeline with emphasis on dynamic stitching. Master's thesis, University of Oslo, Department of Informatics, May 2013.

[6] Mikkel Næss. Efficient implementation and processing of a real-time panorama video pipeline with emphasis on color correction. Master's thesis, University of Oslo, Department of Informatics, May 2013.

[7] Marius Tennøe. Efficient implementation and processing of a real-time panorama video pipeline with emphasis on background substraction. Master's thesis, University of Oslo, Department of Informatics, May 2013.

[8] Ragnar Langseth. Implementation of a distributed real-time video panorama pipeline for creating high quality virtual views. Master's thesis, Department of Informatics, University of Oslo, May 2014.

[9] Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Marius Tennøe, Espen Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad, Asgeir Mortensen, Ragnar Langseth, Sigurd Ljødal, Øystein Landsverk, Carsten Griwodz, Pål Halvorsen, Magnus Stenhaug, and Dag Johansen. Bagadus: An integrated real-time system for soccer analytics. *ACM Transactions on Multimedia Computing, Communications and Applications (TOMCCAP)*, 10(1s):14:1--14:21, January 2014.

[10] Douglas E Comer, David Gries, Michael C Mulder, Allen Tucker, A Joe Turner, Paul R Young, and Peter J Denning. Computing as a discipline. *Communications of the ACM*, 32(1):9–23, 1989.

[11] Vamsidhar Reddy Gaddam, Ragnar Langseth, Sigurd Ljødal, Pierre Gurdjos, Vincent Charvillat, Carsten Griwodz, and Pål Halvorsen. Interactive zoom and panning from live panoramic video. In *Proceedings of Network and Operating System Support on Digital Audio and Video Workshop (NOSSDAV)*, page 19. ACM, 2014.

[12] Tromsø IL. http://www.til.no. Accessed: 2014-06-11.

[13] Simen Sægrov, Alexander Eichhorn, Jørgen Emerslund, Håkon Kvale Stensland, Carsten Griwodz, Dag Johansen, and Pål Halvorsen. Bagadus: An integrated system for soccer analysis (demo). In *Proceedings of the International Conference on Distributed Smart Cameras (ICDSC)*, October 2012.

[14] Pål Halvorsen, Simen Sægrov, Asgeir Mortensen, David K.C. Kristensen, Alexander Eichhorn, Magnus Stenhaug, Stian Dahl, Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Carsten Griwodz, and Dag Johansen. Bagadus: An integrated system for arena sports analytics - a soccer case study. In *Proceedings of the ACM Multimedia Systems (MMSys)*, pages 48--59, March 2013.

[15] Marius Tennøe, Espen Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad, Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Dag Johansen, Carsten Griwodz, and Pål Halvorsen. Efficient implementation and processing of a real-time panorama video pipeline. In *Proceedings of the IEEE International Symposium on Multimedia (ISM)*, pages 76--84, December 2013.

[16] Basler. acA1300-30gc. http://www.baslerweb.com/products/ace.html?model=167. Accessed: 2014-06-04.

[17] Basler. acA2000-50gc. http://www.baslerweb.com/products/ace.html?model=173. Accessed: 2014-06-04.

[18] Lorenz Kellerer, Vamsidhar Reddy Gaddam, Ragnar Langseth, Håkon Kvale Stensland, Carsten Griwodz, Dag Johansen, and Pål Halvorsen. Real-Time HDR Panorama Video. In *Proceedings of ACM Multimedia (ACM MM)*, Orlando, FL, USA, November 2014.

[19] x264. http://www.videolan.org/developers/x264.html. Accessed: 2014-07-28.

[20] Øystein Landsverk. Investigating the performance and applicability of object tracking in a real-time distributed image pipeline. Master's thesis, University of Oslo, Department of Informatics, 2014.

[21] Martin Alexander Wilhelmsen, Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Pål Halvorsen, and Carsten Griwodz. Performance and Application of the NVIDIA NVENC H.264 Encoder. http://on-demand.gputechconf.com/gtc/2014/poster/pdf/P4188_real-time_panorama_video_NVENC.pdf. Accessed: 2014-04-27.

[22] Martin Alexander Wilhelmsen. Real-time interactive cloud applications. Master's thesis, University of Oslo, Department of Informatics, 2014.

[23] Dag Johansen, Magnus Stenhaug, Roger Bruun Asp Hansen, Agnar Christensen, and P-M Hogmo. Muithu: Smaller footprint, potentially larger imprint. In *The proceedings of International Conference on Digital Information Management (ICDIM)*, page 205–214. IEEE, 2012.

[24] Magnus Stenhaug, Yang Yang, Cathal Gurrin, and Dag Johansen. Muithu: A touch-based annotation interface for activity logging in the norwegian premier league. In *MultiMedia Modeling*, page 365–368. Springer, 2014.

[25] TCMalloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`. Accessed 2014-08-10.

[26] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[27] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, page 59–72. ACM, 2007.

[28] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the ACM SIGMOD international conference on Management of data*, page 1029–1040. ACM, 2007.

[29] Apache Hadoop. `http://hadoop.apache.org`. Accessed 2014-08-07.

[30] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, page 13–24. IEEE, 2007.

[31] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, page 260–269. ACM, 2008.

[32] Marc de Kruijf and Karthikeyan Sankaralingam. Mapreduce for the cell be architecture. 2007.

[33] Facebook. `https://www.facebook.com`. Accessed 2014-08-07.

[34] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, et al. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, page 1071–1080. ACM, 2011.

[35] Paul B Beskow, Håvard Espeland, Håkon K Stensland, Preben N Olsen, Ståle Kristoffersen, Espen A Kristiansen, Carsten Griwodz, and Pål Halvorsen. Distributed real-time processing of multimedia data with the p2g framework. *Eurosys,(Poster Session)*, 2011.

[36] Paul B Beskow, Håkon K Stensland, Håvard Espeland, Espen A Kristiansen, Preben N Olsen, Ståle Kristoffersen, Carsten Griwodz, and Pål Halvorsen. Processing of multimedia data using the p2g framework. In *Proceedings of the ACM international conference on Multimedia (ACM MM)*, page 819–820. ACM, 2011.

[37] Zeljko Vrba, Pål Halvorsen, Carsten Griwodz, Paul Beskow, and Dag Johansen. The nornir run-time system for parallel programs using kahn process networks. In *The proceedings of IFIP International Conference on Network and Parallel Computing (NPC)*, page 1–8. IEEE, 2009.

[38] KAHN Gilles. The semantics of a simple language for parallel programming. In *The proceedings of the IFIP Congress in Information Processing*, volume 74, page 471–475, 1974.

[39] Zeljko Vrba, Pål Halvorsen, and Carsten Griwodz. Evaluating the run-time performance of kahn process network implementation techniques on shared-memory multiprocessors. In *Proceedings of the International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, page 639–644. IEEE, 2009.

[40] Zeljko Vrba, Pål Halvorsen, Carsten Griwodz, and Paul Beskow. Kahn process networks are a flexible alternative to mapreduce. In *The proceedings of IEEE International Conference on High Performance Computing and Communications, 2009. (HPCC)*, page 154–162. IEEE, 2009.

[41] Dolphin Interconnect Solutions. `http://www.dolphinics.no/index.html`. Accessed: 2014-07-28.

[42] SISCI API Presentation. `http://www.dolphinics.no/download/SISCI/OPEN_DOC/sisci_api_presentation_IX.pdf`. Accessed: 2014-07-28.

[43] Sockets inteface from the System Interfaces volume of POSIX.1-2008. `http://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_10`. Accessed: 2014-07-28.

[44] InfiniBand Trade Association. `http://www.infinibandta.org/index.php`. Accessed 2014-08-02.

[45] Mellanox Technologies. `http://www.mellanox.com`. Accessed: 2014-07-28.

[46] Intel True Scale Fabric. `http://www.intel.com/content/www/us/en/infiniband/truescale-infiniband.html`. Accessed: 2014-07-28.

[47] Kernel software. MELLANOX Price List as of 21 Jul 2014. `http://www.kernelsoftware.com/products/catalog/mellanox.html`. Accessed 2014-07-28.

[48] Mellanox MIS5022Q-1BFR. `http://www.mellanox.com/page/products_dyn?product_family=89&mtag=is5022`. Accessed 2014-07-28.

[49] Mellanox MCB191A-FCAT. `http://www.mellanox.com/page/products_dyn?product_family=142&mtag=connect_ib`. Accessed 2014-07-28.

[50] Mellanox MC2206130-001. `http://www.mellanox.com/page/products_dyn?product_family=183&mtag=cables_copper`. Accessed: 2014-07-28.

[51] SISCI API documentation. http://ww.dolphinics.no/download/SISCI_DOC/index.html. Accessed 2014-08-10.

[52] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[53] Intel Core i7-2600. http://ark.intel.com/products/52213. Accessed: 2014-06-29.

[54] Intel Core i7-3930K. http://ark.intel.com/products/63697. Accessed: 2014-06-29.

[55] Intel P67 Express Chipset. http://ark.intel.com/products/52810/Intel-BD82P67-PCH. Accessed: 2014-06-29.

[56] Intel X79 Express Chipset. http://ark.intel.com/products/64015/Intel-BD82X79-PCH. Accessed: 2014-06-29.

[57] Dolphin IXH610 PCI Express Gen2 Host Adapter. http://www.dolphinics.no/products/IXH610.html. Accessed: 2014-06-29.

[58] Intel Ethernet Server Adapter I350-T4. http://ark.intel.com/products/59063/Intel-Ethernet-Server-Adapter-I350-T4. Accessed: 2014-06-29.

[59] Intel Ethernet Server Adapter I350-T2. http://ark.intel.com/products/59062/Intel-Ethernet-Server-Adapter-I350-T2. Accessed: 2014-06-29.

[60] NVIDIA Quadro NVS 295. http://www.nvidia.com/object/product_quadro_nvs_295_us.html. Accessed: 2014-06-29.

[61] Circular buffer. http://en.wikipedia.org/wiki/Circular_buffer. Accessed 2014-07-28.

[62] Remote Peer to Peer Made Easy - Dolphin Express IX. http://www.dolphinics.no/download/WHITEPAPERS/Dolphin_Express_IX_Peer_to_Peer_whitepaper.pdf. Accessed: 2014-07-03.

[63] PLX PLE 8747. http://www.plxtech.com/products/expresslane/pex8747. Accessed 2014-08-07.

[64] ASUS P9X79-E WS Review: Xeon meets PLX for 7x. http://www.anandtech.com/show/7613/asus-p9x79e-ws-review. Accessed: 2014-07-12.

[65] Red Hat Reference: Timestamping. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_MRG/2/html/Realtime_Reference_Guide/chap-Realtime_Reference_Guide-Timestamping.html. Accessed 2014-08-10.

[66] Intel Core i7-4960X. http://ark.intel.com/products/77779. Accessed 2014-07-31.

[67] Intel Core i7 4960X (Ivy Bridge E) Review. http://www.anandtech.com/show/7255/intel-core-i7-4960x-ivy-bridge-e-review/2. Accessed 2014-07-31.

[68] Marius Tennøe, Espen Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad, Håkon Kvale Stensland, Pål Halvorsen, and Carsten Griwodz. Real-time panorama video processing using nvidia gpus. In *GPU Technology Conference*, 2013.

[69] H. K. Stensland, V. R. Gaddam, M. Tennø e, E. Helgedagsrud, M. Næ ss, H. K. Alstad, C. Griwodz, P. Halvorsen, and D. Johansen. Processing Panorama Video in Real-Time. *International Journal of Semantic Computing (IJSC)*, 8(2):1–19, 2014.

[70] Svein Arne Pettersen, Dag Johansen, Håvard Johansen, Vegard Berg-Johansen, Vamsidhar Reddy Gaddam, Asgeir Mortensen, Ragnar Langseth, Carsten Griwodz, Håkon Kvale Stensland, and Pål Halvorsen. Soccer video and player position dataset. In *Proceedings of the ACM Multimedia Systems Conference (MMSys)*, page 18–23. ACM, 2014.

[71] Vamsidhar Reddy Gaddam, Ragnar Langseth, Håkon Kvale Stensland, Pierre Gurdjos, Vincent Charvillat, Carsten Griwodz, Dag Johansen, and Pål Halvorsen. Be your own cameraman: real-time support for zooming and panning into stored and live panoramic video. In *Proceedings of the 5th ACM Multimedia Systems Conference (MMSys)*, page 168–171. ACM, 2014.

[72] Asgeir Mortensen, Vamsidhar Reddy Gaddam, Håkon Kvale Stensland, Carsten Griwodz, Dag Johansen, and Pål Halvorsen. Automatic event extraction and video summaries from soccer games. In *Proceedings of the 5th ACM Multimedia Systems Conference*, MMSys '14, page 176–179, New York, NY, USA, 2014. ACM.

[73] Vamsidhar Reddy Gaddam, Carsten Griwodz, and Pål Halvorsen. Automatic exposure for panoramic systems in uncontrolled lighting conditions: a football stadium case study. In *Proceedings of the IS&T/SPIE Electronic Imaging*, page 90120C–90120C. International Society for Optics and Photonics, 2014.